

# **A Sheet Music Organisation System**

## **Final Report**

Submitted for the BSc in Computer Science with Industrial Experience

May 2015

by

**Charlotte Godley**

# Contents

# 1 Introduction

This project focuses on the organisation and display of sheet music. Sheet music refers only to the instructions given to a performer in order to play a composition, and does not include the sound output produced when the piece is performed. Any reference to music from this point onward should be assumed to mean visual sheet music, rather than audio recordings. Whilst Eastern countries and previous eras have used different methods of sheet music notation (**Kaufman**), this project focuses solely on the notation used by western classical music. This type of musical notation is the format most commonly used by orchestras and performers, and in order to understand the problem of organising it, some explanation of the key elements will be required.

Notation of western classical music has used a combination of diastematic and orthographic notation (**RRastall**) since the advent of Gregorian chant around 640AD (**RTaruskin**). The function of this notation falls into two main divisions: the expression of relationship in sound frequency, and the expression of relationship in time, or measure (**oxHistory**).

The representation of pitch or sound frequency of a given note is diastematic, meaning that its relative "highness" or "lowness" in notation provides a visual representation of where its frequency lies. The expression of relationship in time or measure is given by where, in terms of horizontal or measure position, the particular note falls. Further explanation of these symbols and the meaning of notes and pitches will be explained in the problem context portion of this report.

The representations of other parameters in staff notation are normally orthographic, such as indications *p* - meaning piano, or "quiet" - and *pizz* - meaning pizzicato, or "plucked" (**RRastall**). This mechanism is complex in nature, and has a large but finite set of symbols which control every element of the composition. This will be discussed and explained later in this report.

Despite the advantages of digitising music collections notated using this system, the majority of musicians choose to store music physically, using filing cabinets and music cabinets as storage mechanisms (**musicOrganising**). This behaviour can be attributed to the lack of standardisation for browsing and organising digital sheet music. The portable document format (PDF) is the standard digitisation method for documents, which presents a problem for musicians wanting to search by multiple methods because the format does not include meta information about what the document contains (**MusicXMLPresentation**). Another problem in using PDF as a sheet music format is the usability of PDF browsers in musical performances. Whilst the usability problem has largely been solved by tablet applications such as those produced by **forScore** organisation, searching and filtering PDF files is still largely a manual task. Most applications allow for manual input, but with little to no handling automatic information retrieval (**musicReader**). Further alternative software solutions proving this point will be discussed later in the report.

An example beneficiary of an automated solution would be a musical director for an orchestra, who wishes to not only find a specific piece, but find compositions which would work well in a concert schedule. In this case it would be of use to know not only the bibliography of a piece, but also information such as time, speed and instruments, without having to physically look at or memorise each piece. At present, many larger orchestras have a dedicated orchestral librarian, who will handle manual organisation, maintenance and research of the library, as explained and supported by **MusicLibrarian**. The existence and necessity of having someone employed in this position indicates that large libraries take a lot of maintenance, and manual conversion and extraction of meta data to convert a physical library to a digital collection would take a long time.

As such, this project solves an organisation problem by extracting meta data about sheet music automatically, with additional features provided in order to improve the usability and shorten the amount of time needed to create and expand a digital collection of music.

This document discusses the aims and objectives of this project, technical scope and depth of the project, process and method used to produce the solution, and finally a critical evaluation of the project as a whole.

## 2 Aims and Objectives

### 2.1 Project Aim

*The overall aim of the project is to design and develop a sheet music library application, with the ability to organise and view personal sheet music collections, and download sheet music from the internet. Time permitting, it should also be able to generate sound from the sheet music, and import editable music from flat images.*

### 2.2 Primary Objectives

The following objectives are of the highest importance to the project, and are a measure of whether the project has been completed. The technical terms used in this section will be explained in more detail in the problem context.

- **Rendering of sheet music files**  
It will be necessary to render one or more formats of commonly stored sheet music files, as the aim of the project is to enable users to *view* and organise their sheet music collections.
- **Extraction of metadata**  
The project will be required to extract important information from each piece, ranging from the simple nominal data such as title, composer, lyricist, to the more complex notation such as clefs, key signatures and meters used. Automatic extraction of information is considered the most important objective of this project, as this is something that is more often a manual job and will be the most useful to users.
- **Ability to search metadata extracted and auto-generate playlists**  
From the extracted metadata it should be possible to search the catalog of music for specific requirements, such as key, clef, meter, time signature. The system should also be capable of generating playlists based on related data, in order to provide what the user might want to know without having to search the database.
- **Connection to online music collections**  
It should also be possible to connect to online music collections, as it would be beneficial to users to be able to search and add to their collections using the same interface.

### 2.3 Secondary Objectives

The secondary objectives are to be completed only if they do not threaten the completion of primary objectives.

- **Audio playback**  
It would be useful to a cross section of users to be able to play music files as sound clips. This enables performers who regularly play with an accompany musician or ensemble to create practice accompaniments from their sheet music, or hear an approximation of how a melody should sound.
- **MusicOCR conversion of images to parseable Music files**  
It would be easier for musicians to merge their physical and virtual music collections for automatic organisation if the solution provided a way to import flat, scanned sheet music into marked-up music files. This is not a primary objective because whilst it would be a useful feature, the research area for MusicOCR is vast and considering it vital to the project may detriment some other more useful and more innovative features.
- **Difficulty Rating**  
In addition to metadata scanning, it would be useful if the system would determine a rating based on the information given of how difficult the piece will be for a performer. This information would help the

user to avoid needing to visually scan the music for an indication of how much practice time the piece will require.

## 3 Background

### 3.1 Problem Context

This problem's main focus is the difficulty of organising classical sheet music, and how this can be made easier by the automatic extraction of key pieces of information. In order to understand what a performer may want to know about a particular piece, it is important to have a brief understanding of the elements of musical notation common to all compositions.

The key element of this form of notation is the staff, as shown in figure ?? . This is a grouping of five horizontal lines, with each line or space in the staff indicating a different sound pitch, a term meaning the relative "highness" or "lowness" of the sound (**classroom** ).

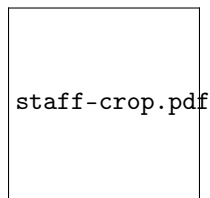


Figure 1: A blank staff

This staff is divided by bar lines, vertical lines delineating grouped units of sound and silence (formally referred to as notes and rests), which provides an indication of the unit's relationship in time by its juxtaposition to other groupings in the composition. These groupings are called measures or bars, with each bar having a variable maximum of notes and rests.

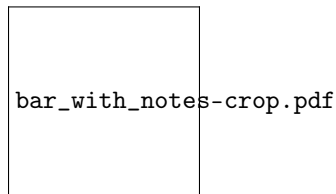


Figure 2: A bar containing three notes and one rest

#### 3.1.1 Clefs

In the system of staff notation, sound frequencies, or pitches, are denoted by letters A-G - after each cycle of the letter names, the next pitch above it will be the start of a new cycle. The cycles are often split by octaves, a term meaning eight pitches, for example A to A or E to E.

In order to provide a link between the lines and spaces of a staff and the pitch names, a clef symbol is necessary.

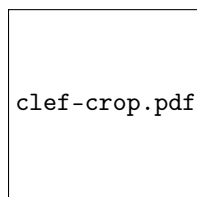


Figure 3: A staff with a treble clef

Each clef symbol denotes a different pitch name - figure ?? shows a G. The center around which this symbol is drawn - in figure ??, the second line from the bottom of the staff - indicates that this line or space will be known as the pitch name denoted by the symbol. From this the reader can infer all other pitches by counting through the letters of the cyclic octave system, so in figure ??, the space above becomes an A, and the space below becomes an F.

This symbol is important to a musician as different clefs are used to position the majority of the pitches in a piece on the staff, as this makes it easier to read. From this a performer can infer the average range of a piece, and predict whether this will be comfortable for the performer's chosen instrument or voice.

### 3.1.2 Keys

A second important indication to the player is the key, denoted by a key signature.

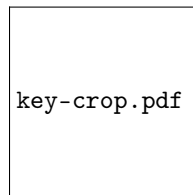


Figure 4: A staff with a key signature

The key signature is a collection of symbols at the beginning of the piece which indicate which pitches should be raised by half pitches, and which should be lowered. Raised pitches are called sharps, indicated by the # symbol, whilst lowered pitches are called flats, indicated by the  $\flat$  symbol. Each key, which has a letter name and key type (which can either be "major" or "minor"), has a different combination of flats or sharps. All pieces have a key, regardless of whether there are any flats or symbols notated here. In the case where there is no key signature, the piece is in one of two most common keys - C major, or A minor.

This is a useful piece of notation to a musician as pieces in less common keys, such as C# major or F# major, may prove more difficult for the user to perform, and therefore they may want to filter out pieces in these particular keys. Similarly, in the case of singers, a singer's range may sit comfortably in one or two keys and they would perhaps want to find pieces in only these keys.

### 3.1.3 Meter

The third symbol denoted at the beginning of a measure is the meter or time signature, displayed as two numerals positioned like a mathematical fraction.

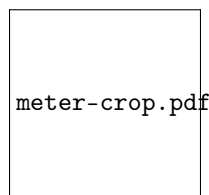


Figure 5: A staff with a 2/4 time signature, or meter

The upper number of a meter symbol indicates the amount of beats in the bar. A beat simply refers to a note or rest, and the type of beat is indicated by the lower number. In the case of figure ??, 2/4 indicates a measure will contain 2 quarter notes. The most common time signature is 4/4, which for this reason is usually denoted with a C in place of the fraction, meaning "Common time".

This information is important as it tells the performer how the rhythm and beat of the piece should be felt, counted and performed, and is useful for searching purposes as different meters give the piece a different feeling, dictating the sort of occasion this piece would accompany.

For example, 2/4 is commonly used for march pieces, and 3/4 is commonly used for waltzes and dance pieces.

### 3.1.4 Tempo

The speed of a particular piece, or the tempo, is indicated by an equation.

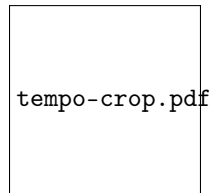


Figure 6: A staff with tempo marking

The equation above the staff in figure ?? indicates that the piece should be played at 60 beats per minute. The symbol dictating the sort of beat per minute depends on the time signature, here a crotchet (or quarter note) is given as the piece is in 4/4 time. Sometimes, this will be accompanied by a text direction to indicate speed or style, such as Andante, indicating a walking speed.

This indication would prove a useful identifier as pieces of different tempos provide variation in performance lists, so a concert organiser may want to find pieces with a variety of tempos.

### 3.1.5 Further metadata

Aside from these symbols, there are some items of textual information useful to the user.

The first of these would be the parts in the piece and their transpositions. A part refers to a grouping of measures given to one performer, as shown in figure ?. "Part" used in a general sense usually refers to the names given to the left, in this example, "Clarinet in B $\flat$ " and "Flute".

Figure 7: Two separate parts in one score

Parts would be relevant as a particular group of instrumentalists may need parts that fit their instruments. If this is not the case for a given piece, however, a part written for a different instrument, for example, the Alto Saxophone rather than the Tenor Horn, may be compatible with the instrument anyway, if the transposition matches the instruments together.

An instrument which has a transposition means that, whilst most instruments would play a note as it is written, a transposing instrument will automatically sound the note in a different key, as described earlier, which may raise or lower the sound of the note. For example, the note C played on an Alto Saxophone will sound as an E $\flat$ , because it is in the key of E $\flat$  major.

Further to this, the user would want to know the piece's title, and names of publishers, composers, arrangers and lyricists of the work, commonly known as the bibliography of a piece (MIR). Further to the composer name, it may be useful to know the date of composition as an indication of the era in which the piece was composed, such as Classical/Baroque/Romantic, though this would not always be written on the sheet music so may need to be researched using the Internet.



### 3.1.6 Sight Reading and Difficulty Grading

This portion of the problem context relates to the difficulty grading secondary objective. It is important to understand why this would be useful to a musician, and as such what follows explains some more specific technical terms which a musician may use to assess pieces of music.

Difficulty grading is a subconscious act which takes place during the initial phase of learning a piece of music. This phase is often referred to as "sight reading", a term meaning that the performer has had little to no practice in performing the sheet music (**sightreading**). It is referred to as sight reading because the musician can only perform symbols as they occur in the music, or as the reader sees the symbols. They may not be familiar with where and when these symbols occur, how and at what speed to perform certain note patterns are to be played, or what keys are to be applied at specific points in the music. The lack of practice can contribute to the performer playing the piece hesitantly, or with mistakes.

After more practice, a performer will generally gain muscle memory of certain repeated note sequences and be more aware of what changes are coming up without needing to read the music. In some instances a performer may memorise a piece of music through repeated practice without consciously intending it.

In order to gauge whether it will be worth sight reading a piece of music, the performer will often visually and subconsciously assess the music for difficulty. This is generally subjective, depending on the level of confidence and ability of the performer, and can change depending on the chosen instrument. However, there are some elements of music which affect this grading for all musicians.

Examples are a note or rest of a very short duration in a fast tempo, particularly in sequence with other short notes. An example is given in figure ??, wherein the notes are a 16th of the duration of a quarter note, and the tempo is 180 quarter notes per minute, indicating 3 quarter notes should be played per second.

Figure 8: A rapid sequence of demi semi quavers, (a quarter of an eighth note in duration)

This is considered difficult because it is hard to ensure that the duration of each note is precise, and in sequence it can be difficult to ensure that the performer can fit all of the notes into the pattern with precision. Often these sequences can cause a performer to blur the notes together, meaning that one note is indistinguishable from another, and may not even be heard if the performer has not timed the note correctly.

Complex rhythms, a term referring to the different durations of a particular sequence of music, can also cause difficulties, particularly for instruments like the piano in which the player has to play in polyphony. Polyphony means multiple sounds or lines of note sequences. In the case of a piano player, it is common for pieces to present cross rhythms in which the left hand and right hand are not playing the same rhythm. Precise timing is required to get the rhythms synchronised with each other.

An example of this is figure ?. Here the right hand plays two quarter notes, whilst the left hand plays three eighth notes. The three above the eighth notes indicates that the pianist must fit the three notes into the same time sequence as one quarter note, known as a "triplet" because the player must fit three notes into a time space where there would normally only be two.

Figure 9: A cross rhythm

The developer has assessed this area with some detail, and results from a survey of other musicians with varying level of ability as well as chosen instruments, with some analysis of other factors are given in the appendix.

## 3.2 Comparison of Technologies

### 3.2.1 Programming Language

This project could be developed with a variety of programming languages, as displayed in table ?? . These three languages have been selected on a subjective basis due to the developer's programming language experience.

Language	Speed of development	Developer's Knowledge	Most recent use
C#	Fast	A lot	2nd year
C++	Slow	Average	2nd year
Python	Fast	A lot	In constant use for over a year

Table 1: Table of languages considered

The first language in consideration is C#. C# is mostly used on Windows due to the main compiler being closed source and developed by Microsoft, but with some platform independence due to the Mono Project, which is feature-complete to C# 10 (**MonoDev** ). There are several other possible tools to create applications for other operating systems using C#, such as Xamarin Studio, but the developer has not developed any applications with C# for use on multiple operating systems.

In terms of speed of development, the developer considers that the language syntax is reasonably intuitive and consistent. C# is statically typed, which means the code is less likely to contain bugs at run time, but reduces the flexibility during development.

Finally, due to the language being mostly for Windows and largely being closed, C# has a lower percentage of Open Source projects on the popular online source code hosting service Github (**Redmonk** ). Whilst this is not necessarily important to development at this stage, the developer intends to Open Source the project and if there is a lower percentage of repositories for this language, that could potentially indicate a lower number of contributors.

The second language in consideration is C++. Whilst C++ is arguably the closest to native code and therefore the language which will be the best for cross platform development, the syntax and memory management issues of C++ mean that development can be slower, particularly when the developer does not feel as competent using this language.

The third language in consideration is Python. Python's syntax is the closest of the three considerations to English or pseudocode, and thus requires the least amount of key strokes and makes the project more likely to attract a wealth of less technical contributors, in particular musicians, as the language should be easier to learn how to use.

Python is dynamically typed, which makes it more flexible to work with but could also introduce more problems at run time if types are not handled properly by the developer.

Python also has a higher percentage of new Github repositories than the other languages in consideration (**Redmonk** ). Again, this does not indicate that Python is the best language for Open Source as that hypothesis would be hard to objectively prove, but it does potentially indicate a higher number of available developers with knowledge of Python, so a large available community of potential developers for when the project is open sourced.

Lastly, there are many current projects written in Python in the area of music research (**pmus** ), which means the project will be easier to integrate and communicate with other projects or build upon the work of others without needing to port software to Python.

It is for these reasons that Python has been chosen as the development language. Beyond the selection of Python, it is important to discuss and consider which version to use, as Python 3 was introduced in 2008, but Python 2.7 continues to be maintained and was updated to include many of the backwards compatible

features in 2010. Whilst this seems an obvious choice as Python 3 is the latest version, many projects still have issues updating and using Python 3 as it was deliberately not backwards compatible (**Foundation2** ).

Upon due consideration, Python 3 has been selected. This decision has been made because the project does not require any libraries which are not available for Python 3, and therefore the developer should make every effort to keep the project up to date with the latest version.

### 3.2.2 File format

The project will require at least one default format for it to process music, which needs to have detailed information about what the score contains. This is necessary for information extraction and for the generation of readable sheet music. Table ?? describes the options considered.

Format	Purpose
new format	this project only
muscx	MuseScore notation
SIB	Sibelius notation
MusicXML	sharing music between software

Table 2: A table showing the different file formats considered

The first option is to create an entirely new format. This would mean the file format was designed to the requirements of the project and therefore would be entirely customisable and extensible. However, this would require further design into how the file would be structured, loaded and unloaded. It would also not be implemented as standard to software which composes music, and as such would either require manual file creation or require writing a conversion script to and from other formats. This project is created with the intention of organising, not composing music, therefore creating an entirely new format would be considered a drain on the developer's time when other formats which are more standardised could be implemented without conversion scripts.

The next two options, muscx and SIB files, are formats used by the open source composition software MuseScore (**MuseTour** ), and the world's most popular proprietary composition software, Sibelius (**avid** ). Using either or both of these files would mean the majority of users would be able to use the application. However, both formats would couple this project with those particular packages, when users could still choose other software to write music with. Furthermore, the formats are specifically designed for those software packages and may have nuances which make development for this project more difficult. Additionally, Sibelius is proprietary so borrowing their file format may cause copyright issues.

The final option is MusicXML, a file format intended for sharing and archiving the world's sheet music (**mxml** ). This particular format is used by a wide variety of software packages (**mxml** ) and is included in the formats usable by both MuseScore (**MuseTour** ) and Sibelius (**avid** ). Using MusicXML would avoid coupling the format with a program and would not require manual creation or import of current music files.

Aligning the project with an open format like this will make the project a better renderer, as the project will be designed to handle MusicXML more effectively than other packages which are designed to use their own format by default, then import or export to MusicXML. This method of storing and loading data is used by MuseScore, and comes with several issues and problems as well as inconsistencies, as documented in their issue tracker (**mscoreBugTracker** ).

However, this particular format was designed by a third party, Make Music, who have a vested interest in the file format and its structure as they produce Finale, another popular music editing package (**mxmlSoft** ). This means that the design aims of the file format have a particular alignment to that platform, and will not necessarily be logical to work with.

Using MusicXML also means that there will be a technical challenge of learning how to use and understand the format, which may affect the development time adversely if the developer does not pick up enough initial knowledge to design the system effectively.

MusicXML has been selected as the file format for the project because it is already included in many of the standard composition software packages.

### 3.3 Comparison of Technologies for Rendering Music

Figure ?? shows the flow of data into and out of the system in order to produce a working renderer. Each stage of this process will be discussed in detail in the following sub sections.

Figure 10: A flow diagram describing the process of rendering sheet music from XML

#### 3.3.1 XML verification Considerations

Before XML is parsed for information in the flow chart in figure ??, the file is by default validated using the DTD defined in the XML file, which can optionally be switched off by the developer. This confirms that XML is valid before starting to parse a file which could be corrupt, but the speed at which files will parse is greatly reduced according to the speed of the user's internet connection.

Furthermore, if the user is browsing their own music collection, it should not be necessary for the user to be connected to the internet.

Due to speed and functionality considerations, the choice has been made that the XML parser algorithm will not verify XML before beginning to parse it. Given that most musicXML will be produced automatically by other programs, it is unlikely files opened by the project will be corrupt, though necessary steps will be taken to avoid this causing a problem in the program.

#### 3.3.2 Libraries for parsing XML for information

In order to extract information from each tag in an XML file, there are two potential built in methods to choose from.

The first, known as DOM or Document Object Model, loads the entire XML file into memory and provides methods to search the loaded file for specified tags (**PythonDom**). The developer has used the Python DOM library before in various industrial projects, and believes it is cumbersome to manipulate data in this way. Furthermore, this project is focussing on rendering the information rather than rendering it with precise formatting, and many software packages implant musicXML files with very complex formatting information which may or may not be necessary (**MusicXMLPresentation**).

The second option is using an api called the Simple API for XML (SAX). In this method, the library will load the XML file tag by tag, and connect to developer-defined call backs when specified things occur in the file, for example a new tag or piece of data inside tags, or the closing of an old tag (**PythonSax**).

SAX is easier to work with as functionality can iteratively be built up by creating handlers for each tag, and is better for memory management as only tags which are necessary to the project will have any effect on the program or program memory. For these reasons, this method has been selected.

#### 3.3.3 Algorithms for display and storage of XML information

For the rendering of sheet music, the system must in some way manipulate the data extracted from the XML file into a visual output.

The usual method of formatting an XML file would be applying an XSL stylesheet, which is usually the method for rendering XML files in a web browser. This is not an option in MusicXML because the notation is too complicated and requires too many symbols, whilst XSL stylesheets are usually used for images and text representations. Sheet music is somewhere in between the two, and thus this option cannot be selected.

The second option would be to create or reuse a converter script from XML to the output format, meaning the output would be generated at the same time as the input. However, this couples the system with MusicXML and would slow down development if it were required to use a different additional input, or decisions were made to change the output format. Furthermore, the Big O representation of this type of script to any output format is  $O(n^2)$ , which is not a highly optimised algorithm.

The final option is to create a converter script which would parse the MusicXML and create a hierarchy of objects. This reduces coupling directly to MusicXML as new formats of both input and output would only need a converter script to create object hierarchy or from the hierarchy to the output format. The big O representation of this particular method is  $O(n)$ , a noticeable improvement on the converter script option.

However, whilst this reduces coupling, a technical challenge is created using this method in that the object structure needs careful planning to cater for the structure of MusicXML and the structure or method of output, which, with little knowledge of the input or output format, is difficult to achieve.

Parsing to objects has been selected as the method of choice in order to avoid coupling and create an extensible system.

### 3.3.4 Rendering System

Given the decision made in section 3.3.3, the system must take the object hierarchy and transform it to readable sheet music. The user should be able to pan around the sheet music and zoom in and out of it to view specific details.

This could be achieved using a new system, with the output going directly to the render window using different glyphs and fonts extracted from their relevant classes.

However, the functionality of panning and zooming using this system may be difficult to optimise, as both could possibly require running the system each time the user provides input.

Furthermore, the conversion of even basic sheet music to a readable format would require a high level of precision and complexity. The time put into creating a new rendering system would be considered unnecessary as this is a process that has been covered by many different applications (like MuseScore (**MuseTour**), Finale (**mxml**), Sibelius (**avid**) and Lilypond (**Lilypond**)). Lastly, the process of debugging whether the symbols are correct would require visual checking and would be difficult to debug automatically.

It would be possible to alleviate the panning and zooming problem by converting the collection of symbols to an image or PDF file and using a built in image rendering library to display this output, such as wxPython (**WX**). However, this method still involves the process of creating an algorithm which has already been covered and adopted by many parties, and incurs problems with visual debugging.

The final option is to output the structure to a formatted file, and have an external rendering system parse this formatted file. Lilypond is a language and system developed to typeset the highest quality sheet music (**Lilypond**), which takes an input file and outputs a PDF or image. An example of some lilypond code is given in figure ??, with the processing output from Lilypond given in ??, and the resulting pdf given in figure ?. As this is a language unto itself and has been in development for many years by the Open Source community, this will alleviate the problem of visual debugging - instead, each class can create a formatted Lilypond output based on its attributes and unit tests can automatically confirm that the result is as expected.

```

\version "2.18.2"
\header {
  tagline=""
}
\relative c' {
  c8 d e f g a b c}

```

Figure 11: An example of Lilypond input

```

Processing `~/Users/charlottegodley/PycharmProjects/FYP/
documentation/reports/final/lilypond_example.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `lilypond_example.ps'...
Converting to `./lilypond_example.pdf'...
Success: compilation successfully completed

```

Figure 12: The processing output from Lilypond

Figure 13: The PDF output from Lily-  
pond

There are other options and packages which are part of the  $\text{\LaTeX}$  eco system, another typesetting language. This project typesets documents, rather than sheet music alone. The two major packages are MusiXTeX and ABC (**musixtex**), but as both of these packages are embedded into  $\text{\LaTeX}$  which is unnecessary for this particular project, this may bloat the user's operating system with unnecessary software.

The chosen option for the rendering system is to output formatted files to Lilypond, which will then take the output and generate a PDF.

### 3.4 Comparison of Technologies for Organising Sheet Music

This section discusses the organisation of sheet music. This involves parsing and verifying an XML file as explained in 3.4.1, extracting data according to a metadata model as explained in section 3.4.2, and storing it in order to be searched and manipulated, as explained in section 3.4.3. After this process is completed, the user will want to query the data set which will require some processing of an inputted string, explained in section 3.4.4. The decisions taken over all these areas of organising sheet music will be expanded upon in the sections below.

#### 3.4.1 XML data acquisition algorithm

This objective, like the rendering objective, takes data from XML files in a folder provided by the user. As such, both objectives require decisions on which method of parsing would be most appropriate - DOM or SAX, and verifying or non verifying.

The system will use SAX because there are only a select few tags and collections of information that the database will need to peruse, and SAX means that only these tags will have an effect on the algorithm or memory. The file will not be verified before being parsed in order to avoid the need for internet connection, and to speed up the loading and parsing process.

#### 3.4.2 Metadata Model

In order for the system to scan a file for data, a model of what data to collect will need to be designed and implemented. In normal circumstances there may be a model already described as standard for the subject area. In the case of music and music research, there are many facets and disciplines which have an interest in collecting data about music (**MIR**). For example, a musicologist may want to study patterns, a sound engineer may want to study instrument timbres and transpositions, or an artificial intelligence researcher may want to study the generation of music (**creativeMachines**).

For these reasons, there does not appear to be a standard model, and therefore this project will create a new, general model for music information retrieval. There are two options in consideration for the model of data to collect.

The first model would collect general symbolic data, such as the elements described in the problem context - time signature, meter, clef, key, instruments, transpositions, and bibliography information. This provides data that will be useful to every musician reading the music, gives a reasonable amount of complexity and variation and allows for a wide range of searches. However, this does not provide much information which would be useful to specific instrumentalists. For example, a piece containing a lot of pitch movement or certain sequences of notes may prove to be something one instrumentalist wants to avoid, but for another it may be of no consequence. Furthermore, for the difficulty scanner objective this model would need to be expanded to include note durations and patterns, and apply some semantics to the model to deduce the grading.

The second model would collect data with specifics according to instrument name, and make the parser apply certain rules to deduce what would be useful for each player to know. This provides more precision to the data and has a higher likelihood of being useful data to the user.

However, this would be more technically challenging due to the need for semantics and specific structuring of data in a way that the first option would not, as the data has a lot more variance from piece to piece and part to part.

Additionally, relying on the instrument name to deduce rules to apply may be a problem due to different spellings of names which would mean the scanner would have to prepare for all possible changes. For example, some pieces may use plurals to indicate more than 1 instrument should play 1 part, and some pieces may provide the key as well as the instrument or some may use the default which the scanner would have to deduce, like "Clarinet" or "Clarinet in Bb". Another consideration that the scanner would need to make is language, as some human languages name instruments differently, such as gaita which is bagpipes in Spanish. Selecting human languages in the user interface would not solve this problem, as it is common for composers to produce music which contains the instrument names as they are referred to in their native tongue, and it is not common for there to be an available version in any other language.

The developer has decided to use the first option, but design the system so that the parser could be expanded at a later stage of development to allow for piece difficulty ratings to be implemented.

### 3.4.3 Memory considerations

The metadata algorithm will parse all of the files in a given folder which have the XML extension for the model described in the previous section. It will be necessary in some way to store this information, potentially both physically to avoid unnecessary repeat meta data scans, and in memory to allow the program to manipulate the data.

It would therefore be possible to simply extract the data and store it to an in memory object, which would then be loaded and saved to a serialised Python object file. This would be quick to develop as it only requires using the Python serialisation libraries in collaboration with the XML acquisition libraries, and has a simple system design. However, the algorithms and overall design of the memory object would mimic much of the functionality of a database, so would be considered duplication of other people's work with little technical improvement. Furthermore, any extensions other developers made to improve or change the dataset would have to be written or connect to Python, as this would be the only way to unserialise and understand the outputted file.

The chosen solution therefore is to extract data, then communicate with a database instead of an in memory object. Queries to the database would result in ordered lists of files and tuple data sets which could then be manipulated by the system.

This is preferable to the earlier suggestion because database systems have been worked on by multiple contributors including several well known companies who support open database architectures with funding and developer time (**SQLiteConsortium**). The invested time of other developers and companies means more

time has been spent on optimising the searching and sorting methods, usually through the implementation of a B-Tree file structure (**SQLiteBTree** ). For these reasons using a database would most likely have faster access and search times than an in memory object.

Furthermore, using a database avoids coupling any future improvements to Python, as the most common database structures have well designed APIs for the most popular languages (**MySQLAPI** ).

#### 3.4.4 Query Processing Considerations

The application should allow the user to input a search query and find results which match the input. A user can then select from a list of result options which will render the selected file. In order to deduce what tables and data sets to query based on user input, it will be necessary to structure the query in a certain way.

The first option for achieving this would be to define a querying syntax and provide the user with instructions or a tutorial on how to use this syntax. This allows for a lot of complexity and means that the program will not need to do as much string manipulation and processing.

However, this also makes the program less intuitive to use and could cause users, particularly those who are less technical, to be confused or apathetic to the program if the syntax is too complicated. Furthermore, some searches, such as finding all pieces by Mozart, do not require a high level of complexity as results could be achieved using the word "Mozart" without much context.

Another option would be to allow the user to input anything and try to predict from the input which table to query. This would easily be achievable for text input and for input which has symbols specific to one notation element or another. An example of this would be a time signature, which is the only input which would be represented as a fraction, like 4/4, or perhaps a tempo indication as it would always have an equals symbol in the input. However, this removes the ability to build up complex queries and for some elements, like clef or key, it would not be possible to deduce merely from text what the user is expecting. This could potentially result in the system querying all tables for the data, or result in excluding some elements of meta data in order to avoid a processing overhead.

The selected option therefore will be an amalgamation of the two described options, which will allow for both simple strings, such as 4/4 or quarter=half when defining time signatures and tempos, and for more complex input, such as instrument:clarinet with:clef:alto. This will enable users to search without needing to know or use too much search syntax and is aimed to be intuitive and simple. Instructions for the querying syntax are provided in the user guide in the appendix.

### 3.5 Comparison of Technologies for Importing Online Musical Sources

The project should be able to search online, using various sheet music collections, for new music for the user. This involves selecting sources to implement in the project, and implementing an algorithm for downloading and parsing files from those sources. In addition to these technical considerations, the sheet music on these sources may have different licensing arrangements, so handling terms and conditions of sheet music usage will also need to be considered, as detailed in section 3.5.3.

#### 3.5.1 Sheet Music Sources

The project should be able to communicate with one or more online music catalogs in order to allow the user to expand their collection. Whilst the system should be designed for extendability, at least one source should be integrated to the system to prove functionality.

The source this project focusses on using is **MuseScore Online**, which is a community website created for composers to upload, share and discover compositions using the MuseScore platform (**MuseShare** ).



This has been selected due to the number of files available, the openness of the platform and the well documented API. It will, however, be necessary to manage copyright issues, as pieces published on this website may be published under the license of the composer's choosing and therefore may cause issues with certain types of users, in particular those performing commercially.

### 3.5.2 Searching Algorithm

The APIs for the selected source provides a certain amount of bibliographic information for each piece in the collection. The algorithm for searching this data will need to collect and parse data from the API, and allow the user to search this data in order to download relevant files.

The first option for achieving this would be to search the API using only bibliography information whenever a user enters a query containing requests for bibliography, provide the options to the user then download the file if needed. This would be simple to implement, however it would be slow due to needing repeated connection to the internet. It would also cause an overhead on the server side because of this repeated connection, and users would not be able to search online collections using the advanced methods described in section 3.4.4.

It has therefore been decided that the system will collect all the data from the server about every piece in the catalogue, extract the bibliography information for use later, then download and parse each file for meta data, combine that with the bibliography information and finally, put the data into the database and then delete the XML file.

This would mean the data collected would be the same quality and quantity as locally stored files, and can be searched using the same level of complexity. It also avoids the issue of repeated connections to the API, as this would only require a connection when the database is refreshed, or when a user wants to download a file permanently.

However, this has a bigger overhead due to the need to scan each and every file, and there might be a memory consideration temporarily if the system has downloaded a large body of files in one go and the user's operating system does not have the file storage space.

### 3.5.3 Licensing Considerations

One of the problems with sharing and collecting sheet music is how a piece is licensed. Whilst other catalogs such as the IMSLP (International Music Score Library Project) contain pieces by composers who's music is now in public domain (**imslp**), MuseScore Online has music which is published for a variety of purposes, and therefore the wishes of the composer in terms of sharing and reproducing their music must be taken into consideration.

The first option for handling this would be to avoid it by only downloading files from the server which have the lowest license level, or no license at all. This is easy to implement as it only requires a filter on the API requests, and means that this is a none issue. However, the result is a smaller input set, when some licenses, such as the Creative Commons Non-Commercial license (**cc-nc**) would be useable by application users with certain conditions applied.

The second option is to make the user accept a list of terms before downloading a piece. This covers the licensing issue and has a larger input set. This is the option that has been chosen for implementation.

## 3.6 Comparison of Technologies for Sound Output and Image Input

This section details the input and output secondary objectives which are the ability to create sound from the inputted sheet music, and the ability to extract sheet music information from images.

The extraction of sheet music information from images will require Optical Music Recognition, from here onwards shortened to OMR, to understand the music from an image. This process is a combination of Optical Mark Recognition and Optical Character Recognition (**pakitan** ), and will be further explained in section 3.6.2.

### 3.6.1 Sound output algorithm

The sound output algorithm must, for a given part or selection of parts, output the object structure explained in section 3.3.3 to a MIDI or MP3 file, which can then be played within the program.

It has been decided that each class in the solution will have a method to produce this output, in the same way as the algorithm described for rendering in section 3.3.4, which will be combined into an output file and played.

This creates an extensible architecture, as it would easily be possible to create output methods to other formats in the future.

### 3.6.2 Optical Music Recognition systems

In order to import images or flat files into the chosen file format, it will be necessary for the program to include the ability to apply music optical character recognition to the file, and save the output to MusicXML, which can then be parsed by other parts of the program. This is known as Optical Music Recognition, or OMR.

It would be possible for a new system of input to be produced for converting new imported images into the chosen file format. This would mean the system could be optimised according to the project aims, and provide sufficient technical challenge.

However, this project is concerned with music organisation, not optical music recognition specifically, and as such the project is too large to commit a sufficient amount of time to this particular algorithm in order to make it function as well as other algorithms.

As a reference point, Optical Character Recognition for natural languages has taken many years to develop and perfect, and has been an attractive research area and idea to a wide variety of users (**InternationalConf** ). OMR, or Optical Music Recognition, has been the focus of international research for over three decades, and while numerous achievements have been made, there are still many challenges to be faced before it reaches its full potential (**musicocr** ).

It has therefore been decided that OMR as a topic is too large for this project, and if this goal is included in the project, it will be through communication with other systems, such as Audiveris, an open music scanner (**audiveris** ).

This removes the technical challenge of producing an entirely new system, but adds the challenge of understanding how optical music recognition scanners work, and how they can be integrated with the system, particularly if the third party package is not developed in Python.

## 3.7 Comparison of Technologies for Difficulty Grading

This section of the report details decisions taken to design and implement a system of grading a piece based on metadata collected, in order to give the user an indication of difficulty and from that deduce how much practice the piece will require to perfect the performance.

### 3.7.1 Metadata Model Considerations

The first consideration in this objective is expansion of the original metadata model, as defined in section 3.4.2. There are two options for the sort and extent of expansion this would require.

The first option is to expand the metadata model to include symbols which will make the piece more complex at a general level. This would mean that the data would be useful to a cross section of users, and involve collecting information about note sequences and patterns, rhythms used, and places in which notes are raised or lowered by half pitches.

The second option is to expand the metadata model to include information specific to instruments. For example, the first model does not consider pitch against the range of the instrument, which changes per instrument and could potentially make the piece more difficult.

This problem incurs the same issues of language and spelling as the first metadata model consideration, and for simplicity's sake this objective will use the first more general data option.

### 3.7.2 Rating Algorithm

The algorithm for grading a piece must read in symbols and assess the symbol against a knowledge database for whether the symbol increases or reduces the difficulty of a particular piece.

This could be done in tandem with scanning for metadata scanner. The algorithm would examine each symbol, updating the rating based on what it knows about the symbol, and the musical context in which the symbol resides.

It could also be done after the metadata scanner has collected all the data. The metadata scanner would collect certain note patterns and rhythmic sequences according to whether they match a particular element in the knowledge database, such as "4 semi quavers in sequence".

The first option has been selected, because symbols are considered in context and avoid bulk collections of data which may or may not have an effect on the rating. For example, the given "4 semi quavers in a sequence" has no indication of the speed, which could mean the semi quavers are 1 note every second or 4 notes in 1 second. This context would be found by examining the remaining data, but would be more computationally expensive as this would require a second pass of the data collected, when assessing at metadata scan time would avoid this second iteration of the data.

### 3.7.3 Machine Learning considerations

Both algorithm options rely upon the system having some level of knowledge about what symbols make a piece easier or harder. This objective therefore crosses into the research area of Machine Learning, the process by which a system can learn and evolve its assessment of data according to new information (ACM ).

In order to achieve a grading based on the context of the data, an entirely new machine learning system could be created by the developer. This would make the system specific to music, but machine learning is an area researched by a cross section of developers who have developed algorithms which work for a wide variety of areas, and it is unlikely that the developer could improve on this.

The alternative is for the developer to implement a system developed by an external source. Existing research projects into artificial intelligence relating to teaching a computer to compose music could potentially also be adapted or built upon to this purpose, such as the project by **creativeMachines**. Further to this, there already exists an International Conference on Machine Learning and Music, indicating the area would be too large to implement a new system as an objective, rather than an overall project aim. The developer has therefore chosen to implement an external source rather than attempt their own algorithm.

### 3.8 Summary

Table ?? gives a summary of the decisions laid out in the above subsections.

Section	Subsection	Options	Decision	Reason
3.2 Technologies	3.2.1 Programming Language	Python C++ C#	Python	Easy to use syntax  Dynamic typing  Larger pool of potential open source developers  Cross platform
	3.2.2 File Format	Own Format  SIB  MUSCX  MusicXML	MusicXML	Used in several composition applications  Open format  Technical challenge learning new format
3.3 Technologies for Rendering	3.3.1 XML verification	Verifying  Non-Verifying	Non-Verifying	Faster due to no internet needed  Files generally automatically created by software which should create valid files
	3.3.2 Libraries for XML parsing	DOM  SAX	SAX	Better memory management as info not loaded all at once  Easier to use and iteratively build up functionality
	3.3.3 Algorithms for display and storage of XML	XSL  Converter script to output Converter script to object hierarchy	Object hierarchy	Extensible: future I/O only needs to create converter to/from objects  O(n) speed result  Avoids coupling to I/O
	3.3.4 Rendering System	new system converting objects to render window	output to file to run in Lilypond	Higher quality product as more developers have worked to produce it

		<p>system converting objects to rendered image, then display image</p> <p>convert objects to Lilypond script</p>		<p>Avoids duplication of previous research</p> <p>more precise automated testing through comparison to expected text output for each class</p> <p>Technical challenge of learning new format/language</p>
3.4 Technologies for Organising Sheet Music	3.4.1 XML data acquisition algorithm	<p>DOM</p> <p>SAX</p>	SAX	<p>better memory management</p> <p>Iterative build up of functionality</p>
		<p>Verifying</p> <p>Non-Verifying</p>	Non-Verifying	<p>No internet connection needed</p> <p>Files automatically created by trusted composition software, should be valid files.</p>
	3.4.2 Metadata Model	<p>Specific to instruments</p> <p>General to all instruments</p>	General to all instruments	<p>Standard data set to all instruments, no special filtering required</p> <p>Does not rely on language or spelling of instrument names</p> <p>Provides information which is useful to most musicians</p>
	3.4.3 Memory Considerations	In memory object, output to serialised Python file	SQL-based database	<p>Avoids research duplication</p> <p>Ensures files do not have to be repeatedly scanned if they haven't changed</p>

				Standard database developed by multiple other developers: higher quality + probably faster than memory object developed by 1 developer
	3.4.4 Query Processing Considerations	<p>Define new querying syntax all searches must use</p> <p>Manipulate normal user input to predict type of data required</p> <p>Amalgamation of normal input and querying syntax</p>	Amalgamation of normal input + querying syntax	<p>Simple for simple searches</p> <p>Provides more complexity where necessary</p> <p>More intuitive than learning a long list of commands for one simple query</p>
3.5 Technologies for Importing Online Musical Sources	3.5.2 Searching Algorithm	<p>Search API only, using filters available at API level</p> <p>Download all files on source, parse for data, combine with data from API and delete file, then download PDF when user asks for it</p>	Download all files on source, parse for data, combine with data from API and delete file, then download PDF when user asks for it	<p>same quality of data as local files</p> <p>Requires only 2 network calls: at update time and at download time</p>
	3.5.3 Licensing Considerations	<p>Only view files with lowest license/no license level</p> <p>Allow user to "accept" terms of license then download</p>	Allow user to "accept" terms of license then download	Large set of data but still covers licensing issue
3.6 Technologies for Sound Output and Image Input	3.6.2 Optical Music Recognition systems	<p>Create new algorithm</p> <p>Implement third party system</p>	Implement third party system	<p>Avoids research duplication</p> <p>Technical challenge of implementing other people's work</p>

3.7 Technologies for Difficulty Grading	3.7.1 Metadata Model Considerations	Expand to include symbols which are difficult at general level Expand model to include info specific to instruments	General level	useful to all users  No issues with language or spelling of instrument names
	3.7.2 Rating Algorithm	Do calculation at the time the file is ran through the metadata scanner, assess each symbol in context  Collect bulk data, compare when metadata scan has finished	Do calculation at the time the file is ran through the metadata scanner, assess each symbol in context	Considered in context  No bulk collection of data - only collect what is relevant
	3.7.3 Machine Learning considerations	New algorithm/system created by project  Implement third party system	Implement third party system	Avoid research duplication  Research area too big to complete in the given time frame as a feature of a larger project

Table 3: Table summarising decisions taken throughout the background section

### 3.9 Alternative Solutions

This section discusses other software which is currently available for musicians to organise and view sheet music.

Table ?? shows the alternative options considered in the area of Sheet Music organisation automation. This shows that the closest alternative would be Power Music Pro, though much of the functionality changes slightly according to the platform it has been developed for (**PowerMusic**). Furthermore, Power Music Pro's only improvement on manual organisation is the ability to search by lyric, whilst this project intends to allow for a cross section of other organisation techniques, as explained in the problem context.

Additionally, each of the possible options are released in a commercial environment, with Avid's Photoscore being too expensive for the average user. Seemingly, this project would constitute the only free and Open Source software released for this problem.

A final point to make is that none of these solutions provide a version for Linux based operating systems, whilst this project should be useable on Mac, PC and Linux based operating systems.

Software	Rendering of Sheet Music	Manual Organisation	Automatic Organisation by complex notation	Connection to Online Sources	Audio Playback	OMR	Price	Platform
Avid Scorch	✓	✓	×	✓	✓	×	£1.40 (AvidScorch)	iOS
Power Music Pro/Power Music Mac	✓	✓	partial	✓	✓	×	£49 for PC, £29 for Mac (PowerMusic)	PC & Mac
Avid Photoscore	✓	×	×	×	×	✓	£200 (Pscore)	PC & Mac
Scorcerer	✓	✓	×	×	✓	✓	£15 for iPad version, £26 for Mac and PC (Scorcerer)	iPad, Mac & PC

Table 4: A comparison table of other available software



## 4 Technical Development

### 4.1 User Interface Design

The User interface for this project is designed after looking at the user interfaces used by other music applications. In particular, the developer was inspired by Spotify, shown in figure ??.

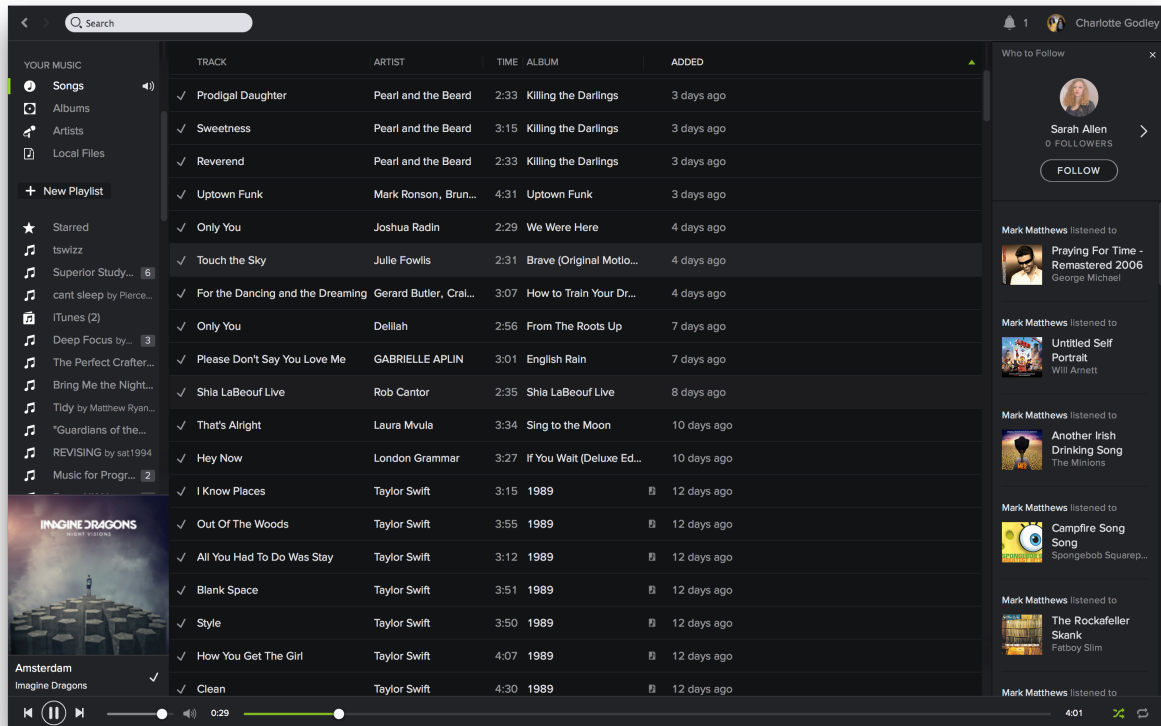


Figure 14: Spotify user interface

#### 4.1.1 Main Display

Figure 15: Main User interface of the project - design

Figure ?? shows the main view of the application, as mocked up before being developed and implemented. The developer took this design, and the others which are in the appendix and produced a prototype using QtDesigner. This prototype is shown in figure ??, updated to show details about the sheet music displayed in the PDF window.

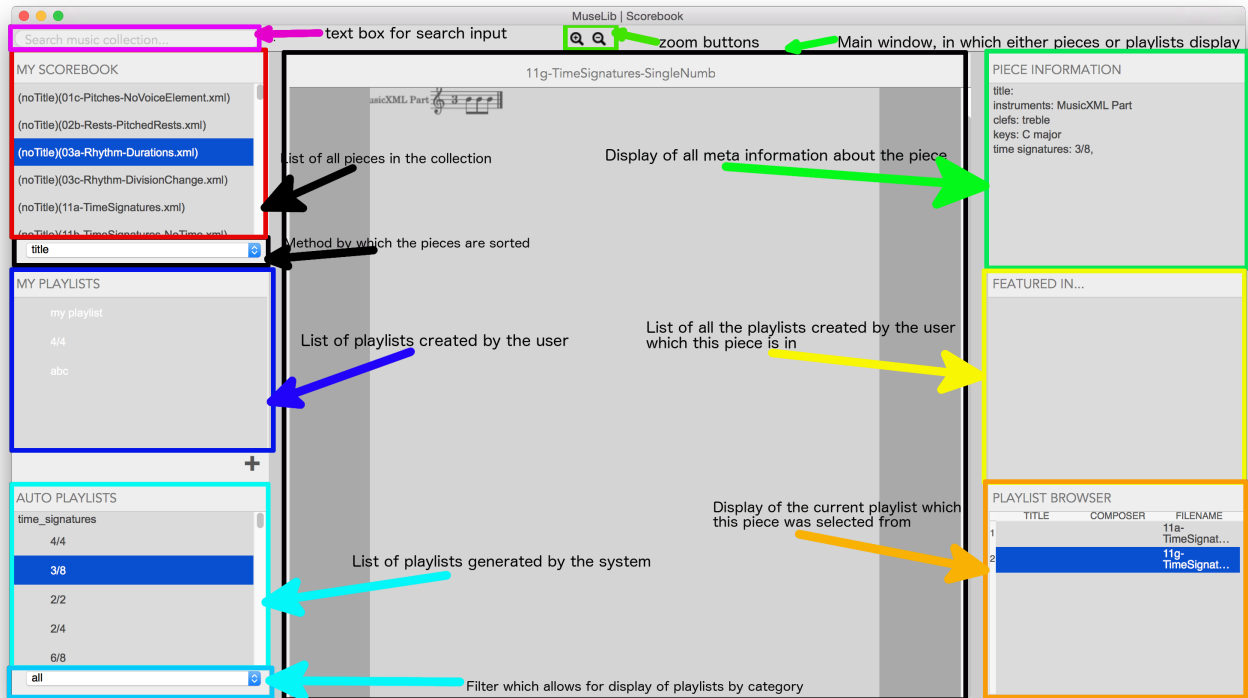


Figure 16: Main User interface of the project - implementation

All of the frames of the user interface are styled using a CSS stylesheet, so that the developer could customise everything with precision. The use of CSS also enabled the developer to easily create new themes. The remaining screenshots with an explanation of how different windows link together and screenshots of the additional themes are given in the user guide which is in the appendix.

## 4.2 User Testing

The developer was aware that this user interface was designed by a software developer who is also a musician. This is a problem because it is difficult for the developer to appreciate whether the interface is too technical for the target demographic. It was the intention of the developer to do user testing of this interface to see how musicians interacted with it and whether any changes needed to be made to make it more usable before implementing it. Due to time constraints and platform difficulties this could not be performed, but the developer intends to evaluate the user interface with real users and modify it based on the feedback before releasing the application fully.

## 4.3 Test Design and System Testing

### 4.3.1 Test Driven Development

The project was developed using Test Driven Development (TDD). Test Driven Development is an Agile software development methodology which utilises the rule that a line of code should not be written unless there is a failing automated test (TDD). This methodology has been chosen because the nature of the notation of music means that attention must be paid to how and with what symbol every element is notated, and Test

Driven Development will significantly improve the quality of the software by closely integrating testing with the development process.

As such, the test plan is the same as the feature list provided in the appendix, as tests were developed at the point of introducing each new feature. Each class and file containing tests was designed to be a self contained unit testing the smallest possible details, such as an accent being added to a measure correctly, or a note's pitch being created with a particular note name or octave number.

### 4.3.2 Unit Testing

Unit tests for this project were created on an objective by objective basis using the standard Python unit tests testing suite. The developer's IDE of choice, Pycharm, has built in support for this module, with an intuitive interface for running tests in a given file and production of readable output of what happened in each tests, as shown in figure ??.

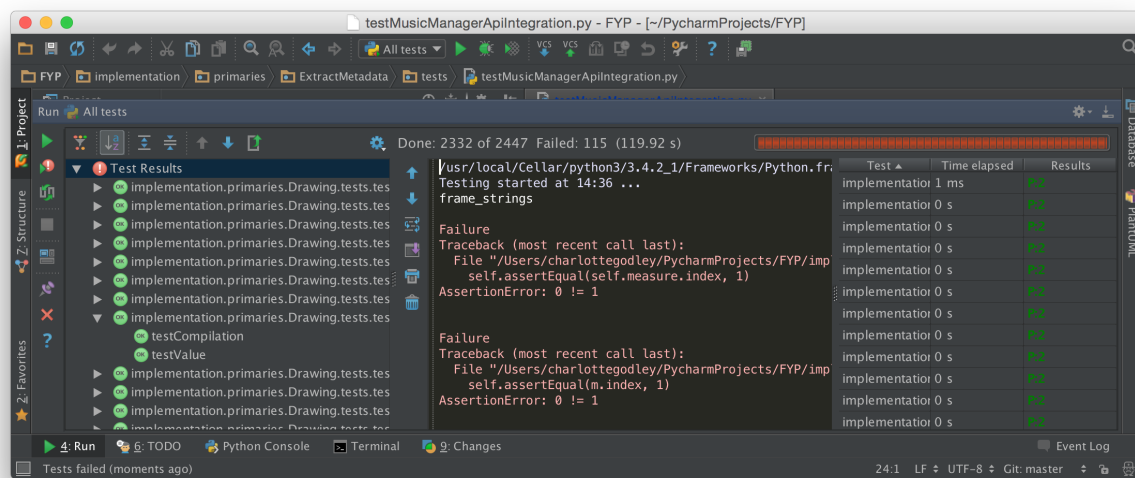


Figure 17: An example of testing integration in PyCharm

The interface also includes the ability to create scripts which discover and run tests on folder, file, class and method level, as shown in figure ??.

The developer used this feature to create scripts which ran all of the tests for specific areas of each objective, as well as the objective itself. This enabled the developer to quickly check new features had not affected the status of previous features.

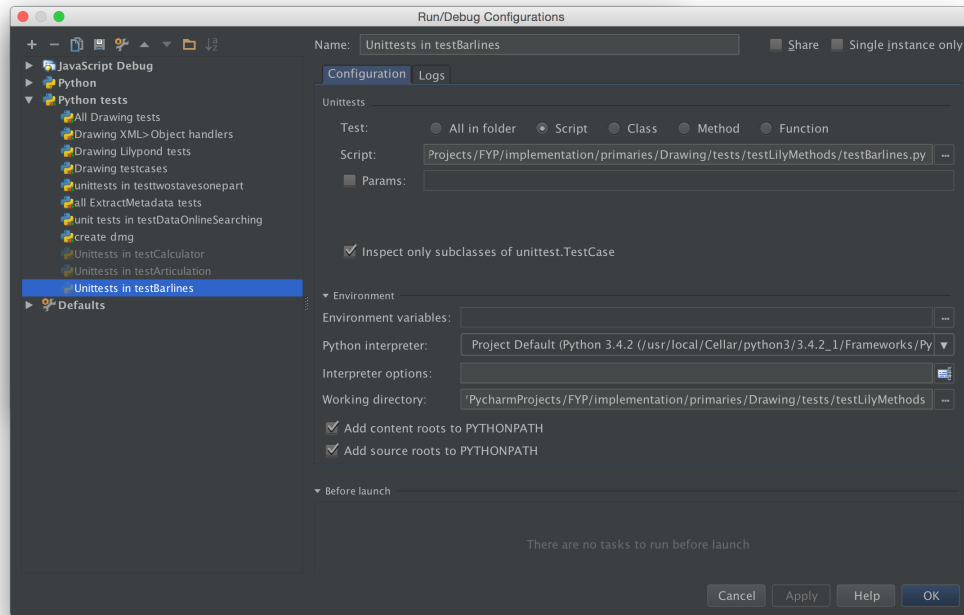


Figure 18: An example of script creation in PyCharm

### 4.3.3 Unit Testing the rendering system

In the initial phase of developing the rendering system, the SAX parser was tested at base functionality level, with checks that new tags, new data and closing tags modified various lists which handler methods used to extract information. As new functions and tags were handled, more and more tests were developed for every tag, termed handler tests. These tests included combinations of tags and combinations of tag attributes, as well as testing tags in exclusion from other tags. When new features were introduced the developer would run the full set of unit tests before committing the code to source control, in order to ensure that new features did not cause regressions or changes in previously developed code.

When the object extraction algorithm portion of the rendering system was completed, the developer created unit tests for the individual symbol classes to confirm their ToLily method worked according to what was expected in the Lilypond documentation.

Once the individual symbols were tested, the developer created tests to check the classes containing the symbols concatenated the strings of output in the correct order. As every possible value and change to each symbol had been tested at symbol class level, this portion of testing was not as extensive and only included tests to confirm ordering of notes, directions and expressions were valid where they had any effect on each other. For example, it would be unnecessary to test a note with every possible dynamic marking, but it was necessary to test a dynamic did not occur before a note.

### 4.3.4 Development of Test cases

In order to test the system more extensively, the developer created several MusicXML test cases. A full list of the test cases used is given in the appendix, as well as a sample file. In order to produce the test cases the developer used the composition software MuseScore, and systematically went through each category of symbols provided producing every possible type of symbol for that category. These were saved as individual files, titled by each category of symbol. The developer then produced unit tests which checked the object

hierarchy when the parser loaded the file. Each test and test method confirmed that the objects were in the right place within the piece, part and measure, and that each object had the appropriate values.

Once the Lilypond output section of the objective was completed, the developer reused the test cases to check their Lilypond output. Initially, the tests were primitive, only confirming that a pdf file with the same name had been created. Later, the developer produced an expected output Lilypond file and checked the output of the test against the expected output.

#### 4.3.5 Use of Third Party Test cases

As there was no official test suite available for MusicXML, the Lilypond project released an unofficial collection of music xml files for testing purposes (**LilypondTestcase**). Each file tests individual elements of music xml, sometimes in collaboration with other areas. In total the test suite contains 131 files and represents the most comprehensive test suite currently available openly.

The developer ran the rendering system against each test case in the suite. Initially, the developer ran them individually, visually checking the output against the images in the documentation for the test suite, and then moving test cases which passed to a separate folder from the remaining test cases. Later, an automated script was created to attempt to speed up the process, though visual checking of the output was still necessary as no Lilypond files were provided with the xml files.

Due to time constraints, the developer prioritised failures according to severity, which had three levels. The first level was failure to produce a Lilypond output due to an exception in the program. The second level was failure to produce a pdf output due to the Lilypond output being invalid. The third and final level was failure to produce a pdf output which matched the expected output. Each failure in any sense was reported using the issue tracker on Github, with an indication of the filename which failed as well as the reason for the failure. An example of such an issue report is given in figure ??.

The developer ensured that all test cases were above the first two levels of severity, and then made a decision on how important the symbol was in the context of its usage in western music, as well as the time remaining to work on the other two primary objectives.

At the point of stopping work on the rendering system, 65% of the test cases in this suite passed and produced the correct graphical output.

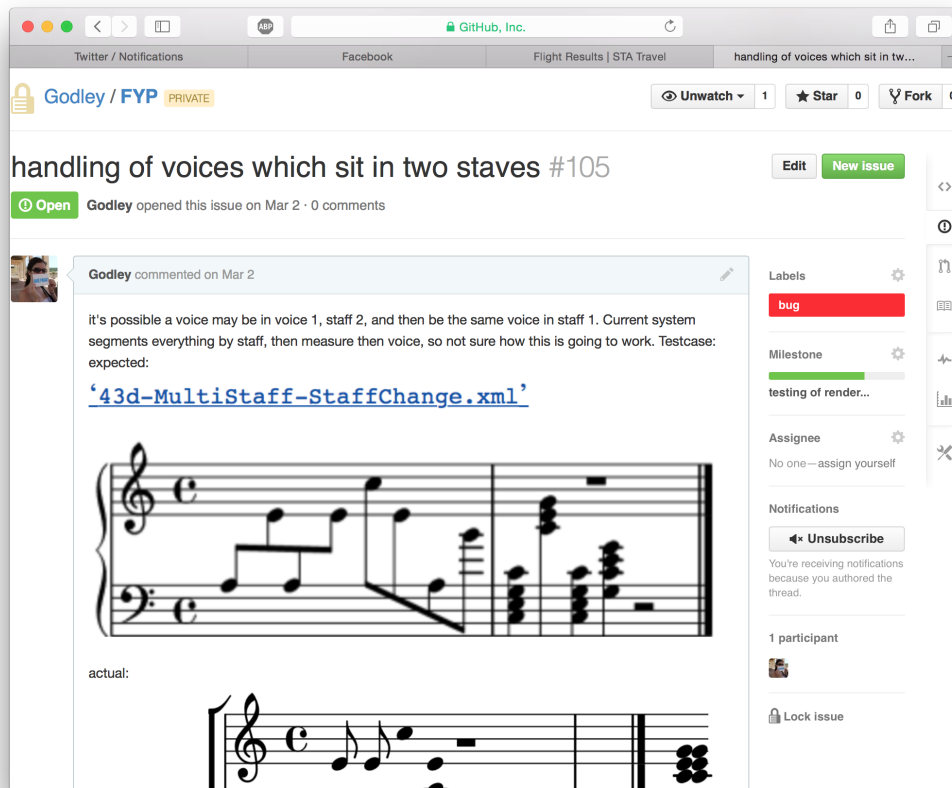


Figure 19: An example issue report

#### 4.3.6 Unit testing the metadata and API objectives

Unit tests were written for the other two objectives in a similar way to the rendering system.

For the metadata scanning objective, the developer tested each element of the system in exclusion from each other. This divided the process into four classes, the folder browser, the unzipper, the data layer, and the metadata parser, as visualised in the class diagram in figure ???. One main class, the music manager, handles the connections between each element. Like the rendering system, this meant that the tests for combining the elements together did not need to be so complicated as the functionality of each element had been tested rigourously.

For the online collections objective, tests were created at source API class level, then API manager level, and finally the API manager was tested when integrated with the music manager class.

### 4.4 System Design

#### 4.4.1 Overall Architecture

The flow chart in figure ??? was designed in the initial prototype phase in order to visualise how the objectives would interlink with each other. A UML Use case diagram is also provided in figure ??, though the flow diagram provides a clearer picture of how the system links together. The use case diagram in figure ?? shows that the user has the ability to import, search for and select pieces. Search will then send a message to the

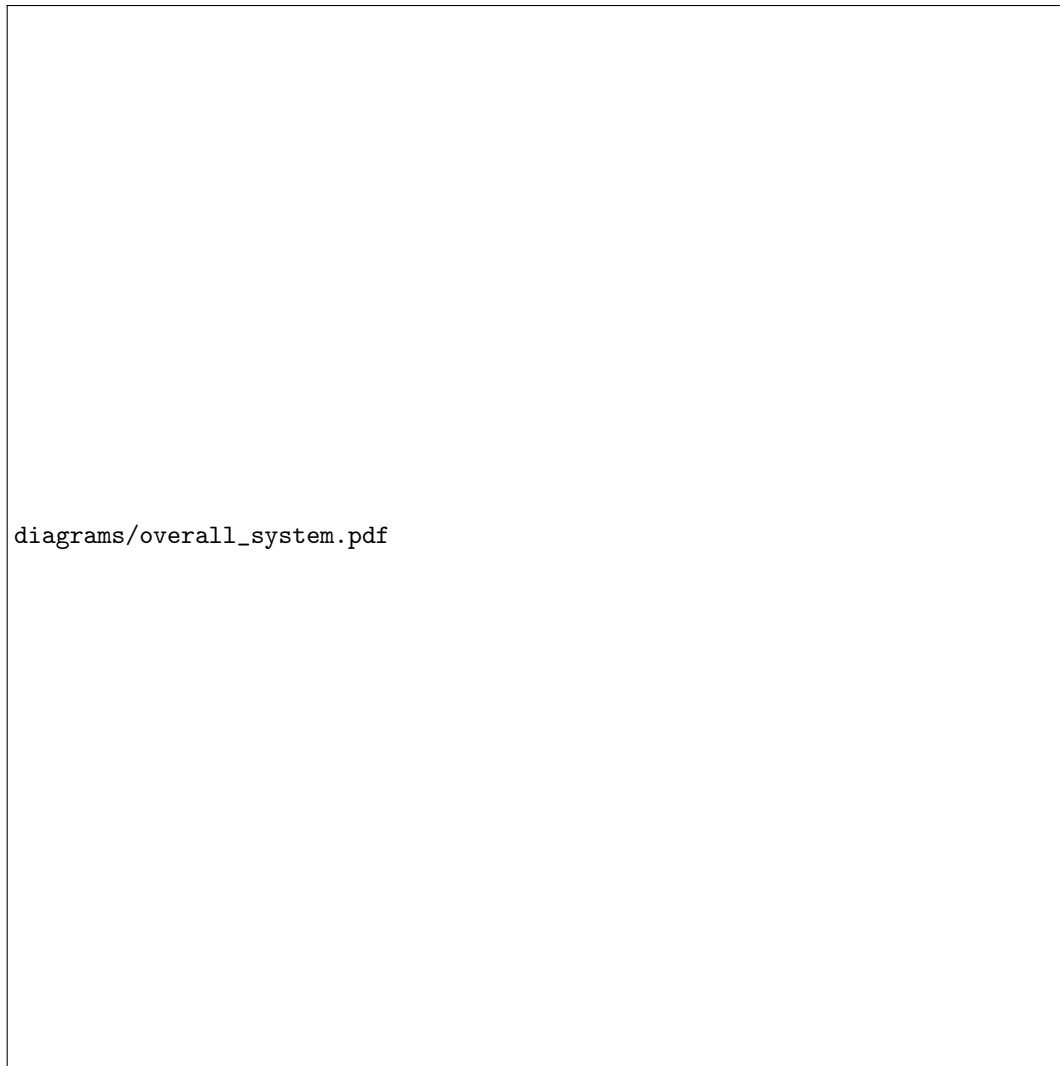


Figure 20: A flow diagram describing the project

music manager, which will process the data and query the database. When the data is returned, it will update the results. The user can then select from these results, either offline or online, and view the piece.

When the user selects an online piece, the use case will request that the music manager find the piece's source and download the file. The music manager will then request that the rendering system display the file. In the case that the file is offline, the system will go straight to the rendering system which will display the file.

In addition to piece features, the user can also create, view and remove playlists. View includes auto generated playlists and the ones that they create themselves using the Create use case.

Figure 21: A use case diagram describing the project

#### 4.4.2 Metadata Scanning system

The flow diagram in ?? refers to applying a metadata scanning system to the folder. This is described in more detail in the flow diagram in figure ??.

Figure 22: A flow diagram describing the meta scanning system

#### 4.4.3 Metadata Class Structure

The metadata scanning objective implements the popular Model View Controller design pattern, which separates the user interface (view) from the program logic (controller) and any databases or information storage (model) (**mvc**). In this instance, as shown in the class diagram in figure ??, the music manager class acts as the controller, which takes input from the application interface. The music manager processes the input and turns it into requests for data, which are sent to the data layer. The data layer communicates with an SQLite file containing several tables, and sends collections of requested data back to the music manager, which updates the user interface with the new information.

The decision was taken to use this pattern in order to avoid coupling to any particular database and to ensure that the objective was properly organised according to functionality. The decision was also influenced by considerations about the future of the project, including expansions to include online data layers or improvements to the UI without affecting program logic.

Figure 23: Metadata and API Class Diagram

#### 4.4.4 Modular Design

The MVC design pattern was complemented by the implementation of the modular design pattern, in which functions and collections of functions are separated into independent blocks or modules (**modular**).

This is shown in the class diagram in figure ?? as the music manager also interacts with other classes such as the folder browser, which handles all functionality involving scanning the given folder for new, old, or zipped files, and the unzipper, which handles the functionality for manipulating zip files.

It is also present in the rendering system explained in detail in section 4.3.5, as each section of the object hierarchy collects its own Lilypond formatting, and then calls upon its child classes to collect theirs, finally collaborating into one complete Lilypond file. In the context of sheet music this is particularly important, as symbolic notation has many options for symbols which may or may not be present, so in program logic they must all act independently of other symbols occurring in the file.

This design was influenced by the decision to use test driven development, as this development methodology aims to test small units of program functionality in an isolated environment (**TDD**). As such, this was easier to achieve if each independent class had one role in the system, with management classes handling the collaboration of these roles without needing to test multiple elements of functionality at once.

This makes the code and overall architecture reusable in different situations. For example, the API manager class sends data collected online back to the Music Manager, which can then communicate with the metadata scanner to extract further information, which avoids code duplication.

#### 4.4.5 Rendering Architecture

The class diagram in figure ?? shows an abstract structure of the sheet music rendering implementation used in this project. This implements a tree, each node of which holds an item containing the notation specific to



that node. Each node object implements the ToLily method, which generates a string of Lilypond formatted output representing itself and then calls each of its child nodes in turn for their outputs, combining them into one string.

A tree was chosen as the object structure in order to give an indication of time, and in order that specific elements could be positioned according to sequential instructions from the MusicXML parser.

Each node in the tree inherits from the Node class. This class gives some generic methods for manipulating and returning its children, which each of the sub classes use in their own specific manipulation methods. Node is different from IndexedNode only in the sense that node contains a list of children, whilst IndexedNode stores them as a dictionary. The indexed node is used in the Staff, Voice, Measure, Part and Piece nodes because each of these classes contains items which have specified keys when they are loaded in using MusicXML, but anything below the voice node does not have an index in MusicXML.

Symbolic classes which add elements to a note or to a measure are added using the methods supplied by measure which contains the logic to decide what to do with the element. The measure handles additions, rather than the note to which the element should be assigned, as some elements must wrap the measure in Lilypond code, some must wrap a note and other more simpler elements add their code to the end of the current measure or note. The logic to decide what to do with the element is easier to achieve at measure level rather than note or direction level so that all options are available.

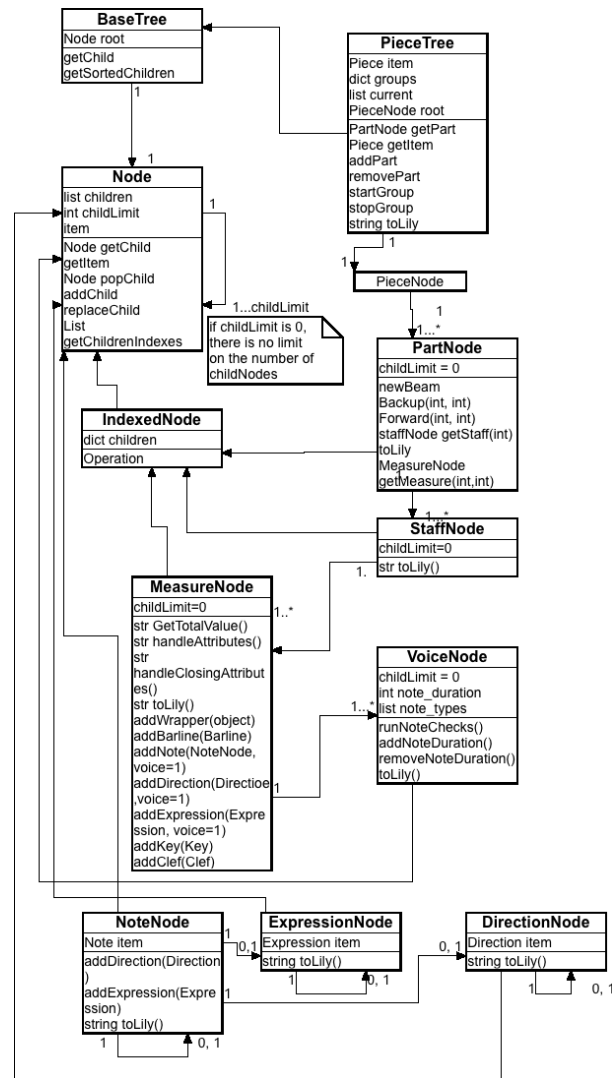


Figure 24: Renderer Class Diagram

#### 4.4.6 Duck Typing

The rendering system visualised by figure ?? implements a feature of dynamic typing known colloquially as Duck Typing. Duck Typing is a method of type inference by which the program assumes that an object is of one type according to the behaviours that it has, not the exact type or attributes it may or may not contain (eric).

In respect to the rendering system, duck typing is applied in the production of output, whereby each class assumes any child classes present will have the ToLily method, call the method and concatenate the results. Using this method of type inference is important to the project as this avoids assumptions about what the music will or will not contain, and instead defines that symbol as suitable for inclusion if it has the ToLily method. Using type inference means that future symbol classes which are implemented only need to provide a ToLily method and be linked into the MusicXML parser in order to be included in the output.

Duck typing can have some negative effects in system design and development. For example, if a class does not have that method this could cause the program to crash without warning. Static type checking used by other languages such as C# and Java would avoid this problem and create a more robust program. However, the use

of test driven development and the generation of test cases have counteracted this risk sufficiently to make the design decision worth taken, resulting in a fluid, experimental and rapid development environment.

#### 4.4.7 Designing for extensibility

The project is designed with a particular aim of extensibility. This affects each area of the system in a different way, but in general, it means that the elements in the system are able to be modified, improved or expanded without requiring changes to the organisation of the system.

An example of this is the API manager, which holds a dictionary of sources, with each index pointing to a class. The API manager will cycle through these sources when its methods are called, meaning that to implement a new API, the developer would simply put a new entry into this dictionary.

Each source implements the API class. To avoid causing problems with classes missing particular methods, the API class will throw a not implemented exception if the sub class does not override the method, in order to indicate to the developer working with a new API that this method is necessary.

#### 4.4.8 Difficulty Rating input from users

When considering the difficulty rating objective for this project, the developer was aware of the subjective nature of sight reading and rating. To assess how this varied from instrument to instrument, the developer produced a survey which was given to a wide range of musicians, as shown in figure ?? . The results of this survey are given in the appendix.

**Final Year Project: A Sheet Music Library**

Duration/tempo questions

**5. Rate the difficulty of a piece...**

	Easy	Medium	Hard
containing semi quavers at quicker than 60BPM	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
containing hemi demi semi quavers at quicker than 60BPM	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
containing demi semi quavers at quicker than 60BPM	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Any other challenging durations?

**6. Difficulty of a piece...**

	Easy	Medium	Hard
with a tempo of > than 120BPM	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
with a tempo of < than 40 BPM	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Any other challenging tempo markings?

**7. Rate the difficulty of a piece...**

	Easy	Medium	Hard
in 4/4	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
in 5/4	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
in 9/8	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Are there any other meters you rate as particularly easy or hard?

Figure 25: An online survey on piece difficulty

## 4.5 System Implementation

### 4.5.1 Development Methodology

The project was implemented using a developer informed process. In this process, the developer would review the project's status for bugs, issues and features which had yet to be implemented and note this using the Github Issue Tracker. When the developer began working with the code, they would select an issue based on the priorities of the project, develop tests for this issue and finally, develop the production code and close the issue.

Issue tracking and reporting was used so that the developer could think and reflect on areas of the system which needed improvement, modification or implementation. The methodology worked well in the context of a year long project in which development was not continuous because it ensured that when development was paused, the developer could see instantly what needed to be done or what was being worked on before the pause to work on something else.

This adds the benefit that in the future, when this project may be worked upon by more than one developer, new developers can see things which have yet to be implemented and choose which features are most relevant or doable by their own standards, as well as discuss or contest previous issues in the system.

### 4.5.2 Challenges in the rendering system

The renderer in this project was designed around two open formats - MusicXML and Lilypond. Whilst every effort was made to avoid coupling with either format, designing the system to work seamlessly with both formats was difficult, and many problems were encountered structuring the object format.

Initially, the project used a simpler object structure in which measures contained a list of items contained within that measure. The decision was taken to use lists of items loaded sequentially because elements belonging to each measure in MusicXML have no unique identifiers which could be used to indicate at which point they occurred within that measure, so objects had to be loaded and stored sequentially. An example is given in listing ??, with the visual representation given in figure ??.

Listing 1: Dynamic followed by note followed by note followed by dynamic

```
1 <direction>
2 <direction-type>
3   <dynamics><pp></dynamics>
4 </direction-type>
5 </direction>
6 <note>
7   <pitch><step>C</step><octave>4</octave></pitch>
8   <duration>1</duration>
9   <voice>1</voice>
10  <type>quarter</type>
11 </note>
12 <note>
13   <pitch><step>C</step><octave>4</octave></pitch>
14   <duration>1</duration>
15   <voice>1</voice>
16   <type>quarter</type>
17 </note>
18 <direction>
19 <direction-type>
20   <dynamics><ppp></dynamics>
21 </direction-type>
22 </direction>
```

Figure 26: Example as represented in sheet music

It was then discovered that Lilypond dictates that dynamics and some other elements had to occur directly after the note to which they should be assigned (**ExpressionLilypond** ). The above listing would look like figure ?? in Lilypond.

```
\version "2.18.2"
\header {
  tagline=""
}
\relative c' {
  c4\pp c4\ppp }
```

Figure 27: MusicXML example written in Lilypond

The system was affected because some musicXML files, such as the one given in listing ??, contain directions before a note has occurred within the measure, which would cause the Lilypond output to be invalid. The structure had to be redesigned based on this information in order to classify notes as first class objects, for which other elements would be indexed according to which note they occurred after. Directions and expressions needed to be split into two lists, in order to ensure that expressions (like dynamics) could be put before directions, but not before the note they should be assigned to.

Later it was discovered that MusicXML allows for navigation through a measure using two specific tags called "forward" and "backward" (**forward** ), which in the original system would require a lot of list manipulation to position elements correctly.

Various other feature inclusions led to the developer redesigning the system around these factors, finally reaching the tree structure described in the design section. This enabled the developer to move objects around according to the tags which next occurred.

Fundamentally, the difference in the design aims for MusicXML and Lilypond is that MusicXML dictates a sheet music file by how it appears, as in the above example, sequentially positioning the elements is how a viewer would see the overall piece. Lilypond, on the other hand, dictates a sheet music file by how it is played - after all, a dynamic without a note would not achieve the desired result as these dictate the volume of a particular element or elements of music.

The change in design of the system was caused by the developer analysing a limited subset of cases for both the input file format and the output format. It may have been a suggestion to extend the research period before development to accommodate this, but a research period long enough to understand the design aims of the formats in enough detail to fully design the structure from the outset would have reduced the development time by a significant amount. Redesigns and refactors were made at late stages of development, and new data structures were considered to solve the problem. The options were analysed according to their various merits in recursive sorting and searching, as well as the links which needed to be made between each symbol or element of music. The late redesign and refactor was implemented after a period of experimentation with new data structures, where a generic tree structure was assessed against a linked list. The test harness highlighted areas which the developer needed to change in order to have a working refactored system. This improved the quality of software and reduced the development time needed to find and fix problems.

However, the negative side of having a test harness so extensive was that changes to production code sometimes required changes to test code, depending on the assumptions of the previous system design. Each test which failed had to be reviewed first for whether it failed because of these assumptions, or whether it was a genuine bug in the production code. Despite this negative aspect, the developer regarded the test harness as a benefit which improved the quality of the software and ensured no detail was missed.

### 4.5.3 Use of external technologies and libraries

For this project, external libraries were used for the graphical user interface and for the database layer. Both were necessary to avoid development duplication and will be discussed in the sections below.

#### User interface libraries

There are several libraries available for creating a graphical user interface in Python, with the most popular ones summarised in table ??.

Library	Benefits	Drawbacks	Cross platform
Qt	Has its own designer program (QtDesigner) and some other third party designer programs  popular amongst many languages  can use CSS to style widgets	Long installation process  no built in PDF viewer	✓
Wx	Built in PDF display  several designers made for it	Designers not as powerful as Qt + all are third party	✓
TKinter	Built into Python itself	No designers No built in PDF/PNG viewers Binds to an old language which is rarely updated	✓
AppKit	Works well on OSX	OSX only	×

Table 5: Table of GUI libraries in Python

In Python there is only one library in popular usage (AppKit) which is locked to one platform, as shown by the table. This library was ignored due to it being locked to one platform.

In the initial phases of GUI prototyping, the developer trialled two of the suggested GUI libraries.

The first trial was TKinter, being as it is built into the Python library and did not require any further installation. TKinter is a library which provides a Python binding to TCL's user interface libraries (**PythonTCL**). The TCL user interface library has not been updated or upgraded for today's image formats, and does not support PDF or PNGs. Any use of either of these formats required installation of other libraries, such as the Python Image Library (PIL) (**PIL**). Furthermore, as discussed by **GuiProgramming** there are very few open, non-commercial options for graphical GUI designers for TKinter, meaning that any interfaces had to be hard coded using python code, rather than creating designed files to be imported.

The second trial was Qt. This library took longer to install, being as PyQt, like TKinter, is only a binding to a lower level C++ library. This means that PyQt cannot be installed in the same way as Wx, using the Python Package Library pip, and for windows meant installing and compiling all sections of PyQt, Qt and SIP which links the two together from source, as no pre compiled installer was available.

In addition to this development time, Qt does not provide a widget for PDF documents, which meant the developer installed a second third party library called Poppler, which is also a C++ library with Python bindings. This library was found to be far easier to use than the Python Image Library used by TKinter.

Whilst this extended the time taken to produce a fully cross platform application installer, Qt provides a fully extensible QtDesigner application, which enables developers to produce their designs, including applying CSS

to widgets, in a graphical user interface which outputs UI xml files.

The xml files are then loaded into Python, meaning that any changes to the files through the designer automatically affect the python binding the next time the developer runs the application.

Furthermore, as Qt is a common library to other languages and is reasonably popular, far more support was found for installing, using and extending Qt than the other libraries the developer tried to use, so the developer decided to use Qt.

Wx was not trialled because the developer felt comfortable using Qt for the reasons given above, and did not want to reduce development time further by trying a new library when Qt functioned as needed.

### Databases and Database Libraries

Table ?? shows the options for databases the developer could have implemented in this project. These three options were selected because all of them are free and have a reasonable amount of documentation for beginners in each database.

Database	Benefits	Drawbacks
MySQL	Open source Widely used/supported	Installation process long
MongoDB	JSON friendly scalable	Low developer experience with NoSQL
SQLite	Single file Portable Library built into Python used by a lot of other applications	some types not supported not very scalable

Table 6: Table of database solutions

The first option, MySQL, was dismissed because this application is not an online solution, and as such, the developer would need to install MySQL and probably have the user install MySQL on their machine when the application was created (**mysql**). MySQL is also a much larger system which is not as suited to a smaller local application.

The second option, MongoDB was dismissed because the developer did not have experience with using NoSQL, the query language used by MongoDB. Furthermore, the recommendations for MongoDB usage are when an application has large amounts of data and needs to scale (**mongod**). It is not expected that there will be a large amount of data when this application is in general usage, or that scalability is a factor unless future extensions use bigger data sets.

The third and final option, SQLite, was selected because of its portability as it only creates one file upon connection. The library for SQLite comes with Python itself and is fully documented in Python's documentation site (**PythonSQLite**). The developer found the library intuitive to use which helped to reduce development time. Furthermore, SQLite is used in a lot of other applications such as iTunes (**sqliteusers**), which means that the sheet music database produced could be more easily integrated and shared with other platforms and environments.

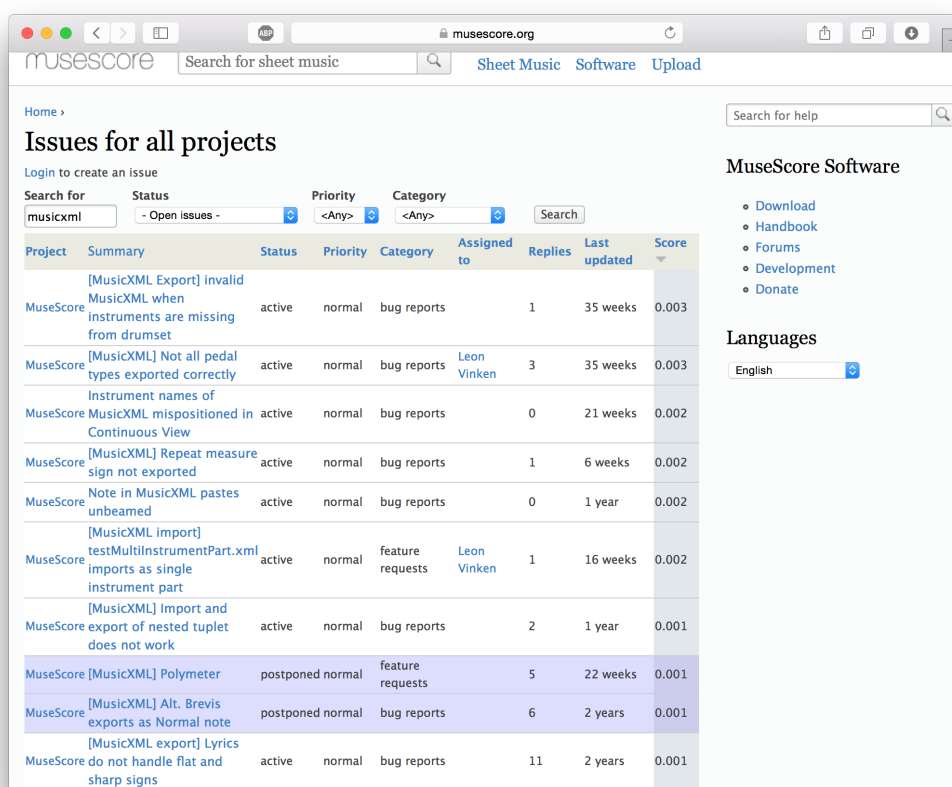
## 5 Evaluation

### 5.1 Project Achievements

All four primary objectives of this project have been implemented, and as such the project is considered a success. The four primary objectives of rendering sheet music, extracting data from the sheet music files provided, processing queries on the data collected and expanding the collection using online sources were set as each contribute a way to automatically organise, expand and view their collection and are achievable, measurable objectives. The sub sections below go into each objective in detail, in order to convey any particular technical challenges which had to be overcome to reach each objective.

#### 5.1.1 Rendering System

This project implements the ability to view sheet music stored as MusicXML files. Composition software such as Finale by **finale** MuseScore by **MuseTour** and Sibelius by **avid** each implement their own file format and provide export and import options to and from MusicXML. Focussing on their own file format means that the input and output of MusicXML is less precise, proven by the number of issues in the MuseScore issue tracker shown in figure ??.



The screenshot shows the MuseScore issue tracker interface. At the top, there's a search bar for sheet music and navigation links for Sheet Music, Software, and Upload. Below this, the page is titled 'Issues for all projects' with a 'Login to create an issue' link. A filter section allows searching by project (musicxml), status (Open issues), priority (<Any>), and category (<Any>). The main table lists various issues, including bugs and feature requests, with columns for Project, Summary, Status, Priority, Category, Assigned to, Replies, Last updated, and Score. On the right, there are links for MuseScore Software (Download, Handbook, Forums, Development, Donate) and a language selector set to English.

Project	Summary	Status	Priority	Category	Assigned to	Replies	Last updated	Score
MuseScore	[MusicXML Export] invalid MusicXML when instruments are missing from drumset	active	normal	bug reports		1	35 weeks	0.003
MuseScore	[MusicXML] Not all pedal types exported correctly	active	normal	bug reports	Leon Vinken	3	35 weeks	0.003
MuseScore	Instrument names of MusicXML mispositioned in Continuous View	active	normal	bug reports		0	21 weeks	0.002
MuseScore	[MusicXML] Repeat measure sign not exported	active	normal	bug reports		1	6 weeks	0.002
MuseScore	Note in MusicXML pastes unbeamed	active	normal	bug reports		0	1 year	0.002
MuseScore	[MusicXML import] testMultiInstrumentPart.xml imports as single instrument part	active	normal	feature requests	Leon Vinken	1	16 weeks	0.002
MuseScore	[MusicXML] Import and export of nested tuplet does not work	active	normal	bug reports		2	1 year	0.001
MuseScore	[MusicXML] Polymeter	postponed	normal	feature requests		5	22 weeks	0.001
MuseScore	[MusicXML] Alt. Brevis exports as Normal note	postponed	normal	bug reports		6	2 years	0.001
MuseScore	[MusicXML export] Lyrics do not handle flat and sharp signs	active	normal	bug reports		11	2 years	0.001

Figure 28: Issue tracker for the MuseScore project

This project focussed on MusicXML, and therefore the input and output of elements should be more precise than the software referenced in the previous paragraph. This is achieved with low coupling to either format, as



the system implements an object structure to manage communications of data into or out of the system.

Technical challenge was overcome in designing the objects used in the system around two separate input and output formats which had very different design aims for how sheet music should be stored and processed. This was achieved in the late stages of development through redesign with the test harness confirming that implementation of the redesign did not introduce any problems.

The classes and scripts for input and output are loosely coupling to other features, meaning that it would be possible for a new developer or new project to take this code and use it in a different context without requiring other objectives and features to make the system work. Using the SAX method of XML parsing means that the renderer can easily be extended and iteratively improved to handle more and more complex XML information.

### **5.1.2 Metadata scanning**

This project implements the ability to organise sheet music through the extraction of information about each piece of sheet music. The information is general in the sense that the data collected is of relevance to the majority of pieces of music and all instruments.

Further to automated scanning, the system allows users to create their own playlists. The search with suggestions system described in section 5.1.3 is modular, so the playlist creator has the same query structure and abilities as the general search function.

The metadata model created for this project was designed so that it would be of the most use to all performers. This design decision was made so that the application would be useful to a cross section of musicians, and with the additional intent of defining a standard model for future projects to implement.

### **5.1.3 Search and Playlist operation**

From the extracted metadata it is possible to search the catalog of music for specific requirements, such as key, clef, meter, time signature. This is achieved using a string of user input which is transformed into queries, which are handled by the data layer.

Suggestions for what the user might be looking for are updated in near-real time and avoid blocking the user from updating their query using multi processing. The challenge overcome here was designing the syntax for a non-technical user, but providing enough scope for complexity to produce a wide range of queries, and was achieved using a mix of language processing and new query syntax.

The system also uses the meta information to produce auto generated playlists which are listed in the user interface. Playlists provide an alternative viewing format to queries, in that it provides a list of pieces which are related to each other, and was made part of the search objective as this is a way in which the data can be used to create an organised user interface.

### **5.1.4 Importing Online APIs**

The application makes it possible to connect to online music collections and search them using the same user interface as searching for local files.

A general API class was implemented in this project to handle communications with networked sheet music collections. This class will be useful to future applications or extensions as new sheet music collections can easily wrap their own APIs in this class, and implement them without needing too many requests or changes at code level.

Whilst the API initially downloads an XML file in order to extract information for every piece in the source, when a user decides to download a file, the API will request PDF files where available. This means that the

renderer does not have to be used, which reduces the processing overhead needed for downloaded files to be displayed.

Furthermore, applying the same model and metadata scanner used on local files to online files makes this feature more useful, as users can search their local collections and online collections using the same queries. This enables users to expand their collections without leaving the application or changing their behaviour.

### 5.1.5 Cross Platform Capabilities

All four of the objectives and the graphical user interface for this project are built to be used on the three most popular operating systems - Mac OSX, Windows 8.1, and Ubuntu (GNU/Linux).

A packaged version of this application has been produced for OSX, with work to produce a Windows installer program ongoing. This was achieved by paying close attention to which libraries were available in Python and avoiding any which did not explicitly state they would function cross platform.

### 5.1.6 Graphical User Interface

All of the objectives and functions of the program are accessed from the same user interface, which provides several views and options to the user depending on how they wish to view and organise their music. As well as automatic organisation, this also provides a create playlist function which allows users to create sub-collections of music which are related to each other, or create set lists for orchestras. The user interface is shown in figure ??, and also in the user guide in the appendix.

## 5.2 Further Work

### 5.2.1 Improved Rendering Capabilities

The current renderer has the capability to handle much of the most common symbols used in sheet music. This varies from the notes themselves to dynamics and articulation, and includes all of the symbols mentioned in the problem context. The rendering system covers all instruments which have parts written using the standard Western Classical sheet music notation, such as Clarinet, Cello, Piano, and Trumpet.

However, this objective in particular took more time than originally intended or expected, and despite the importance of a working renderer in order for a musician to view their sheet music, it was decided that covering the majority of use cases was enough. As such, some specific areas of notation including guitar tablature, drum tablature, lyrics and some notation used in foreign countries and previous generations of western music were ignored. This was due to it becoming obvious that some of these areas would take too much time modifying the current structure, and others were not considered to be important enough to take precedence over the other objectives.

The decision was therefore taken to limit the functionality of the renderer at the advantage of having more time to work on the remaining objectives, but with clear instruction noted for future development on which areas needed to be improved or included in the future. This was achieved using the test cases released by **Lilypond** with visual checking of the output and reporting issues on which test cases failed, and why they were considered failures. It is important to note that the developer ensured that all of these test cases did not cause a program crash, and that all test cases generated some form of PDF output.

### 5.2.2 Creation and release of an Open Rendering Library

This project had baseline requirement of developing a sheet music rendering system in order to achieve the "view sheet music" part of the aim. The baseline requirement was needed because there was no third party

library available for download and installation which would do the same job without extracting the code from a previous project (**pypi**).

It is the intention of the developer to take the current rendering system and produce an independent rendering library, which will be used by the project but also released into the Python Package Index (PyPi), making it available to be downloaded and used by anyone else who needs to use a rendering system.

### 5.2.3 Secondary Objectives

The three secondary objectives were not included in the final project outcome. The decision was taken to ensure that the three primary objectives were all tested thoroughly, and that the application is a rounded and polished product, with packaging for OSX and Windows taking precedence over features which may or may not be finished to the same quality as the four primary objectives.

The first secondary objective, Sound output, was not implemented because the developer felt that it would take time researching the best file format to use for sound output. Furthermore, the developer would need to research and learn to use the libraries needed in Python to produce those files using music symbols. Lilypond supports output to midi using the midi command, but this would tightly couple the sound output objective with the rendering objective. This area was researched however in the sense that the developer planned the structure for producing sound output, which would be the same implementation as the rendering objective, using a method call like ToMidi() or ToMp3.

The second secondary objective, image input, was not implemented because the developer felt the area high risk because of the potential amount of work and research required. Research was put into finding third party OMR libraries like **audiveris** and **openomr** but Audiveris did not appear to have an OSX installer which is the developer's development platform. OpenOMR did not have sufficient documentation for the developer to be able to learn the functionality in time, and both projects are written in Java which means that the project would need to implement some sort of extension to communicate between the platforms.

The third secondary objective, difficulty rating, was not implemented because the developer did not have sufficient time or knowledge of machine learning algorithms. Whilst the developer had a clearer picture of how to implement this objective, using contextual analysis of each symbol against a knowledge database, the developer felt the aspects to be researched had the potential to take too much time understanding how they worked. Research here was done into collecting data about which instruments had difficulty with particular aspects of music, which involved producing a survey which was then given to a wide range of non-technical music. The results of this survey are given in the appendix, and will be used in development of this objective in the future.

The developer will prioritise the second objective, image input, because it is believed that this would be of the most use. Implementation of image input would make the process of importing and organising new music from physical copies completely automatic if the application implemented links to scanners and cameras, which would reduce the time and effort taken to digitise sheet music collections.

### 5.2.4 Porting to Linux based operating systems

The intention of the developer was to create a packaged version of the application for the three major operating systems - Windows, OSX, and Ubuntu.

Time constraints meant that the developer had to prioritise based on usefulness and user count for the chosen demographic, and it was predicted that Windows and Mac OSX would be the most used operating systems by musicians.

Whilst it is expected that the GNU/Linux installer will be relatively simple to create, the developer chose to leave this process to after the demonstration.

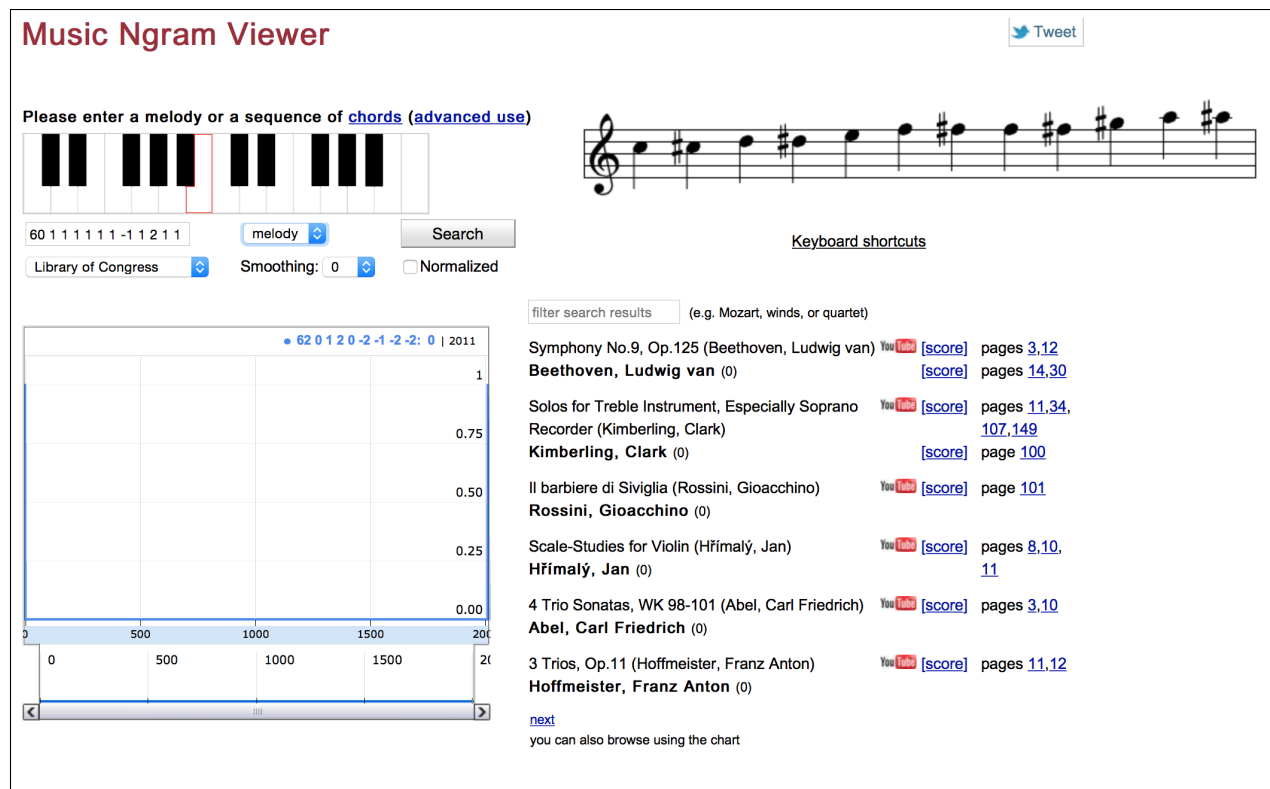


Figure 29: Peachnote Ngram Viewer

## 5.3 Future Developments

### 5.3.1 Porting to Raspbian

It is hoped that a Raspberry Pi compatible version can be created which may involve optimising certain features in order to fit on a smaller operating system.

With this comes potential for new input mechanisms, such as the PiPiano, an add on board which allows users to input music using buttons arranged in a piano keyboard organisation (**pipiano**), and new output mechanisms, such as Sonic Pi, which was created as an educational tool to teach children how to program using music as the inspiration and final output (**sonicpi**).

### 5.3.2 Symbolic Searching

The current project uses text input formatted in a particular manner in order to query the database. It is hoped that in future, a Music Ngram searcher such as **Peachnote** can be used.

The Peachnote viewer is an engine which presents the user with a staff, and allows for input via a virtual piano keyboard interface. It then searches the database it is connected to for any instances the note pattern the user has selected. A screenshot of this interface and the results produced is shown in figure ??.

Using something similar to the Peachnote interface would make the application more musician-friendly, and would mean that novice musicians would not need to know all of the symbol names in order to query the database.

### **5.3.3 Advanced rendering and sound output options**

It is hoped that the project may be expanded so that users can pick and choose what parts and notation are rendered and outputted to sound.

This would enable the system to be used as a teaching tool, whereby teachers and students could choose to simplify the music by displaying only the symbols they have learned. It would also allow the sound output generator to be used to create accompaniment parts for practice purposes.

### **5.3.4 Implementation of paid or subscription APIs**

A further extension of the online API implementation which would be useful is the implementation of new sources which are not open or free.

This would be useful as these sources, such as MusicNotes which implements a pay-per-piece business model, generally contain a larger variety of sheet music, and at a higher quality as they are generally published by the original arranger or composer.

### **5.3.5 Centralisation of APIs**

The current project does all processing of online APIs locally. This could be improved by using a centralised API server, which would communicate with other APIs to access information and scan new files for meta data, which would be stored in a database.

The project would then connect to this API alone, sending formatted query input from the application and receiving suggestions and downloads from the APIs.

This would mean the local user's machine would not have to process XML files which may or may not be requested by the user, and that inclusion of new APIs in the eco system would not require users to update.

### **5.3.6 Online Sharing**

The metadata system currently stores all data to a single SQLite database. This could be expanded in future to offer users the opportunity to upload that data to a centralised online database, in a similar way to the suggested improvement in section 5.3.5.

Users would then be able to browse a larger collection of music from a centralised online point, and request downloads from user created collections. Users who shared their collections would be able to view, accept and reject requests, with an accept resulting in the application uploading the file to the server, sending it to the user requesting the file, and finally, delete the copy on the server.

This functionality would allow users to send files to each other in a more intuitive way than using email or other cloud based services, and wouldn't require much more code or changes to the current system architecture. This may, however, incur problems with licensing which would need to be addressed.

## 6 Conclusion

This project intended to solve the problem of organising sheet music. Sheet music is a visual representation of a piece of music which is given to performers in order to understand how a piece of music is to be performed. The organisation problem has many facets to how it could and should be solved, as some useful information about a piece of music is symbolic and some is bibliographic, some information is general, and some information is specific to each instrument (**MIR**). Automating this process is a feature that previous systems have avoided because the information can be subjective and specific to the context in which the information is needed.

Musicians generally use physical storage methods for sheet music because previous systems did not implement automation of data extraction, and thus whether physical or virtual, the user would have needed to create or use a manual tagging system (**musicOrganising**).

This project is considered a success because it alleviates the manual organisation problem by automatically extracting useful information about pieces of music. It uses this information to provide a platform with different options for viewing and searching collections of sheet music as shown and explained in the user guide, and also implements the ability for users to expand their collection using online sources of music by searching using the same interface as they would their local collections.

An orchestral director could use this project to organise arrangements they have produced using composition software, and expand it by searching for pieces online using the application. Similarly, a composer could use it to organise and showcase the pieces they have created, and more easily find pieces they may have produced in the past without needing to remember the file name.

Whilst the goals of this project were met, the developer learned that the process of rendering sheet music is a more complex process than first expected. The degree of complexity of sheet music notation means that the project does not cover every symbol possible in music composition, but covers enough of the most commonly used symbols to be considered a success, such as dynamics, key signatures, clefs, and a full range of instruments which have parts written for them using the Western Classical notation system.

The developer benefited from having measurable, achievable primary objectives, but with much scope defined by the secondary objectives. The use of test driven development in this project improved the quality of the software and ensured that late design changes did not introduce problems to implemented features.

In the future, it is hoped that more will be done on this project to improve the process of digitising music collections as detailed in the evaluation section. In particular, conversion from images to the format used in this project would make the process from scanning physical copies of sheet music to organising the collection completely automated. Automation of the entire process would mean that the orchestral director would be able to combine their physical collection of music with their virtual collection, reducing space needed to store the physical music library and the time taken to find pieces of music.

The contributions of this project to the field of music information retrieval and organisation are threefold. The first is a rendering system which could be extended and modified depending on new file input and output as decided by future developers, but which is designed to work well with the input and output sources which are currently integrated. The second is a metadata model which is general enough to cover a wide variety of musicians, but with enough symbolic information that it is of more use than the bibliographic and textual data that other systems provide. The third is the project application itself, which provides a usable, extensible application which could integrate new research on a platform which unites the software development community with the music community.

# Appendices

## A Initial class diagram

Figure ?? shows the initial class diagram, which later was changed to a tree. This used a relatively simple implementation of a piece storing a dictionary of parts, indexed by their part ID which is given in MusicXML. Parts then contain a dictionary of measures, which are first indexed by their staff, and then by their measure ID, again given in MusicXML. This construct was extended in the tree structure to include an extra layer between measure and note called "voice", because one part and one staff may still have multiple players using it. In the initial diagram, voice was not considered due to lack of awareness at design time. Here, measure contains a list of items which can either be notes or directions, with all sub classes of either being valid entries into the item list. These are added and accessed directly, which was dropped in later redesigns because it meant that tests had to be changed if any restructure occurred at class level and is generally considered bad practice.

Note may also contain a list of items referring to any accents or expressions which are note-specific. It should be noted that this diagram is simplified, as for example note contains other attributes like Pitch which do not add much to a system design as they are only used and accessed from within the note object.

Figure 30: A class diagram based on the initial mind map

## B Initial User Interface Design

### B.1 Startup Window

On running the application, the user will be presented with the window in figure ?? . This allows the user to create a new collection by selecting a folder where their music files will be stored, or selecting a previously created collection by selecting one of the folders in the list on the left. The user may also choose to remove any of these if they so wish, which will remove the created SQLite file from that folder.

If the user has already created a collection, this window will only display if the user closes the main window, or clicks "new collection" from the file menu in figure ?? . Instead, the application will load the data from the last collection location it was given, in order to avoid showing the user too many popup windows.

Figure 31: Startup window presented when the application opens

### B.2 Main Window

Figure 32: The main GUI

Figure ?? shows the main graphical user interface. The widgets to the left show the scorebook, which is a full organised list of every file in the collection. The drop down box contains title, composer or arranger, and allows the user to reorganise the scorebook according to the filter decision from this drop down.

The next box is the playlists box, which shows the titles of playlists the user has created. These can be removed by highlighting them and clicking the bin button, or viewed by double clicking them which will update the

display as shown in figure ??, or add to them by clicking the add button, which will display the popup box shown in figure ??.

Figure ?? shows the file menu as selected from the application's upper menu. Clicking import will open the popup in figure ??, clicking refresh will force the database to check for new files or the removal of old files within the folder, and clicking new collection will close the window and reopen the startup window. The view menu will present the user with the option of closing or opening any of the widgets currently displayed, whilst the themes menu provides a list of current styles which the user may choose to apply.

Figure 33: File menu

Figure 34: Popup box that opens when the user clicks "import files"

Figure 35: Popup box on click of the "add" button, creating a new playlist

### B.3 Changes to main display when sheet music selected

Figure 36: The main pane when sheet music is selected

The image shown in figure ?? shows the updated main pane of figure ?? when a piece of music has been selected. The smaller windows to the right show an about pane, which displays all information about the piece itself, a "featured in" pane which will list all playlists containing this piece, and a "playlist" pane.

Within the about pane, if a piece of data such as "Saint-Saens" in figure ?? is underlined, it can be clicked which will lead to a playlist of other pieces which contain the same data.

The playlist pane will only display if the piece has been selected for viewing from an existing playlist. That is to say, if this piece were to be clicked from the collection browser shown in figure ??, this window would not be open.

### B.4 Changes to main display when playlist is selected

Figure 37: The main pane of the GUI when a playlist is selected

Figure ?? shows the main pane of figure ?? when a playlist has been selected. This can occur by clicking a manually created playlist from the second pane in figure ??, an auto-generated playlist from the third pane in the same figure, or by clicking on underlined text in the about pane of figure ??.

The button which looks like a pen allows the user to edit the title of the playlist. Further to this, the user may move pieces up or down the playlist by dragging the row up or down the list.

## C Difficulty Grading Survey Results

The following screenshots show the results of the survey given to a selection of musicians in order to assess how they grade pieces of music according to difficulty.

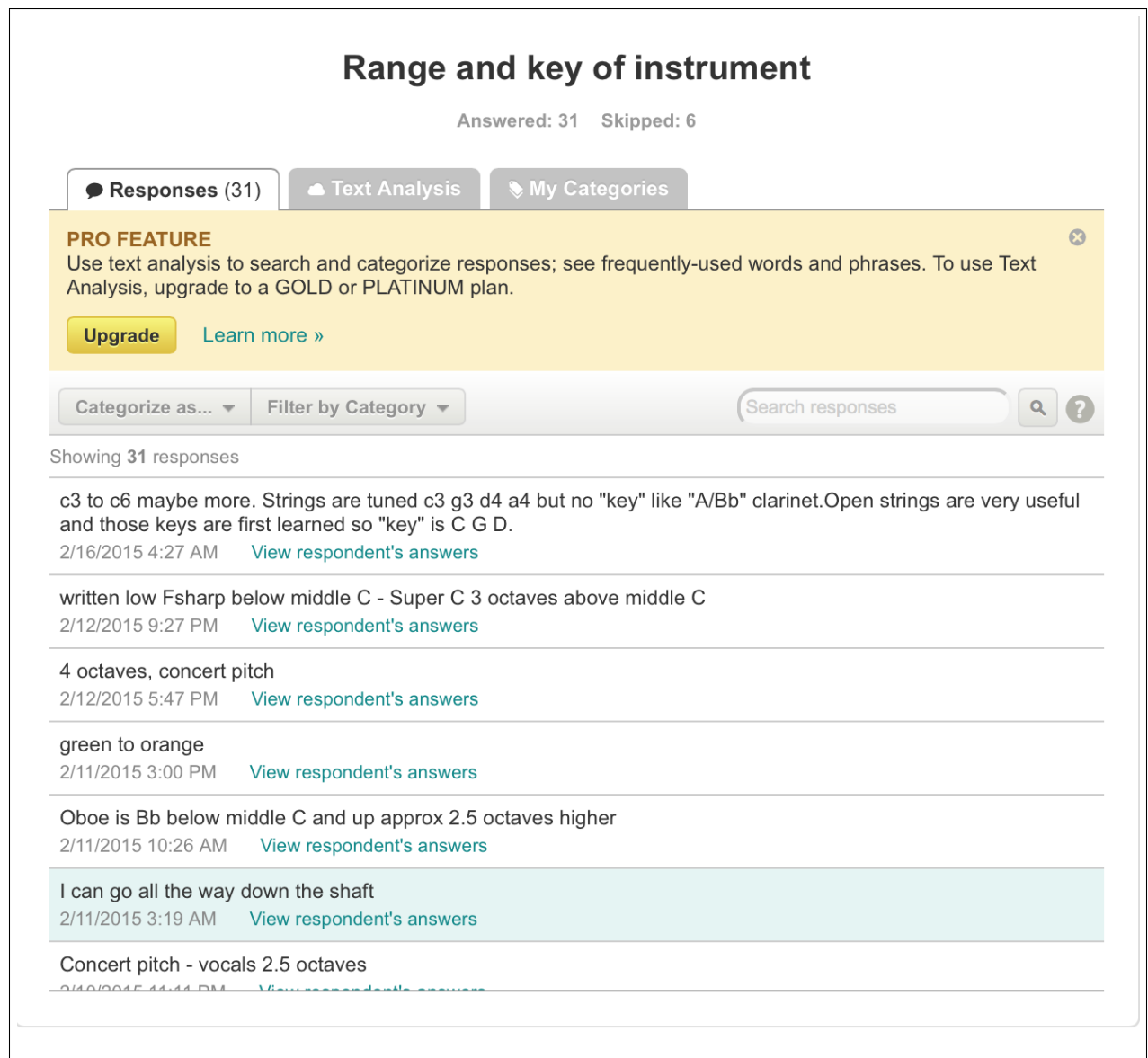


Showing 36 responses		
Clarinet, piano	2/10/2015 10:59 PM	<a href="#">View respondent's answers</a>
Flute and Piano	2/10/2015 10:55 PM	<a href="#">View respondent's answers</a>
Bass Guitar	2/10/2015 10:48 PM	<a href="#">View respondent's answers</a>
piano, flute	2/10/2015 9:59 PM	<a href="#">View respondent's answers</a>
Cello	2/10/2015 9:54 PM	<a href="#">View respondent's answers</a>
Baritone, trombone, saxophone, piano, guitar, clarinet	2/10/2015 9:47 PM	<a href="#">View respondent's answers</a>
Piano	2/10/2015 9:25 PM	<a href="#">View respondent's answers</a>

**Figure 38:** Summary of Instruments each person played

The first question, shown in figure ??, asked the person what instruments he or she plays. The list contains a wide range, which is important as only asking a few instrumentalists would yield very different results.

The next question, shown in figure ??, asked the person what range and key their instruments have. Range is an indication of the number of octaves the instrument is able to produce. This information is useful to know in order to analyse differences, for example instruments which are not of the same family but have a similar range might have similarities later in the survey.

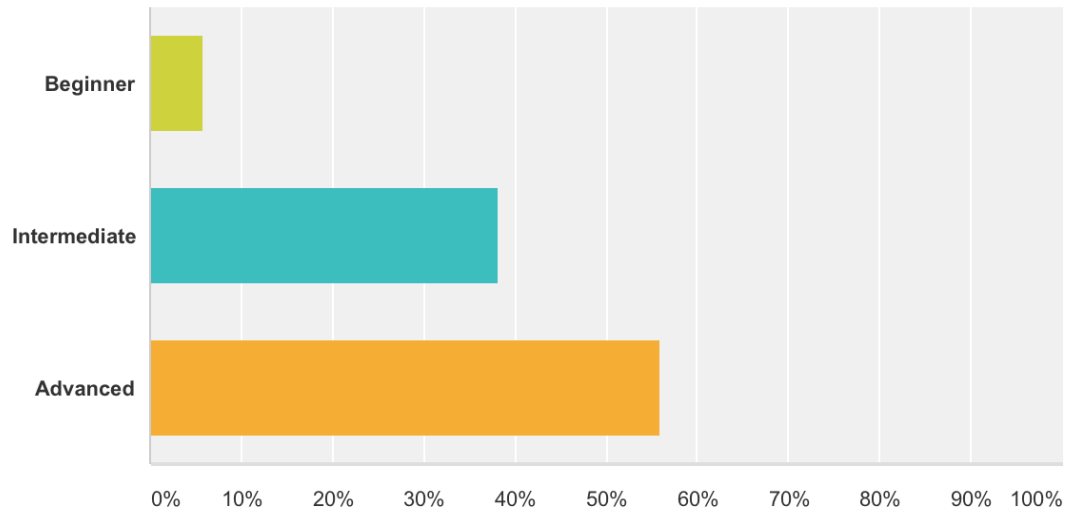


**Figure 39:** Summary of Range/Key

The third question, answers in figure ?? asked the person's ability. "Graded" exams indicate Associated Board of the Royal Schools of Music examinations which give an indication of skill, 1 being the lowest and 8 being the highest.

# What ability level would you rate yourself? If it's different for each instrument, specify in the graded music exam box

Answered: 34 Skipped: 3



Answer Choices	Responses
Beginner	5.88% 2
Intermediate	38.24% 13
Advanced	55.88% 19
Total	34

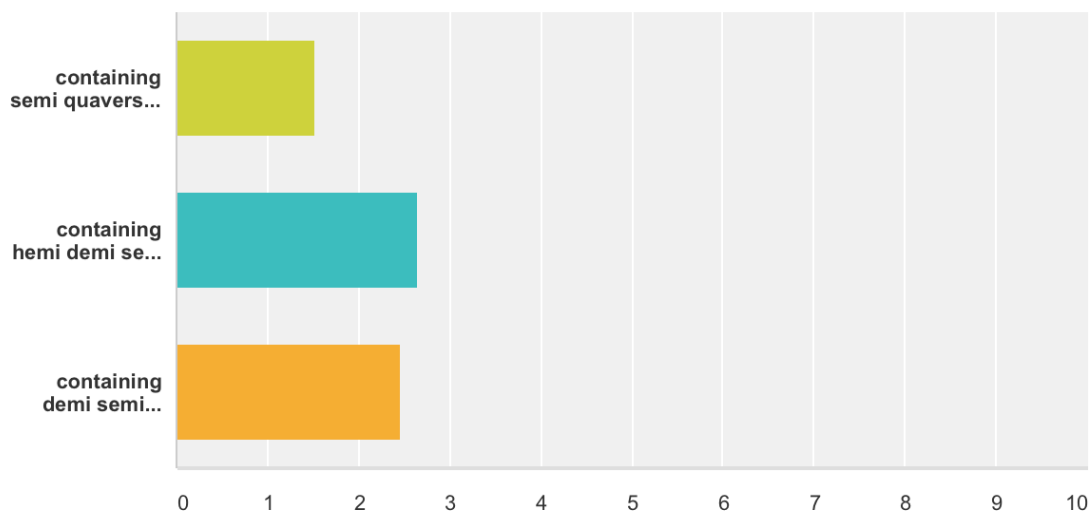
[Comments \(32\)](#)

**Figure 40:** Summary of Abilities

After this, the survey contained a series of questions where the user was asked to rate the difficulty - easy, medium or hard. A comments section was provided in order to give more specifics, particularly for instruments which had a particular reason for the difficulty grading.

## Rate the difficulty of a piece...

Answered: 32 Skipped: 5

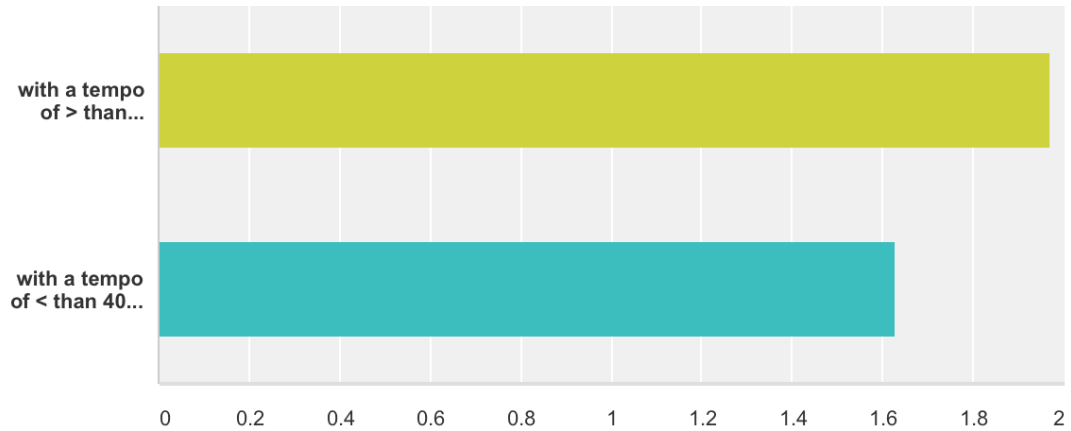


	Easy	Medium	Hard	Total	Weighted Average
containing semi quavers at quicker than 60BPM	54.84% 17	38.71% 12	6.45% 2	31	1.52
containing hemi demi semi quavers at quicker than 60BPM	3.13% 1	28.13% 9	68.75% 22	32	2.66
containing demi semi quavers at quicker than 60BPM	6.25% 2	40.63% 13	53.13% 17	32	2.47

[Comments](#) (11)

## Difficulty of a piece...

Answered: 30 Skipped: 7

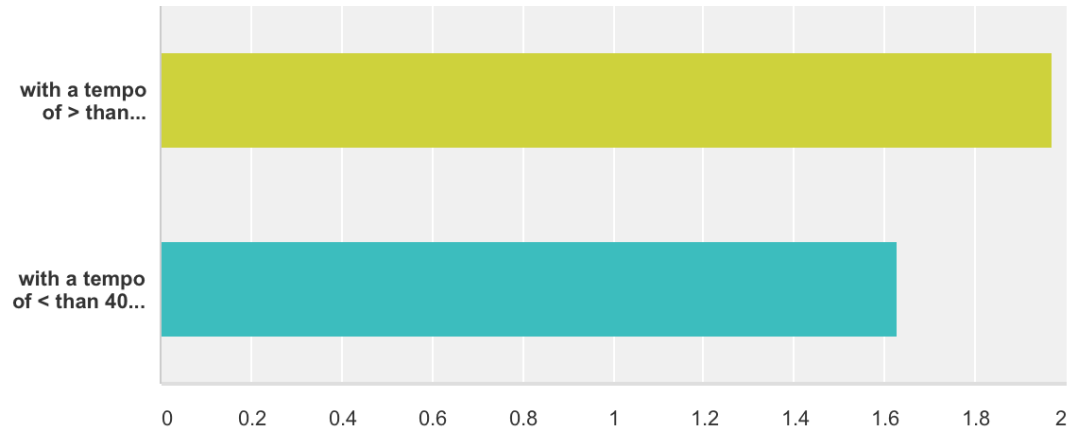


	Easy	Medium	Hard	Total	Weighted Average
with a tempo of > than 120BPM	26.67% 8	50.00% 15	23.33% 7	30	1.97
with a tempo of < than 40 BPM	50.00% 15	36.67% 11	13.33% 4	30	1.63

[Comments](#) (8)

## Difficulty of a piece...

Answered: 30 Skipped: 7

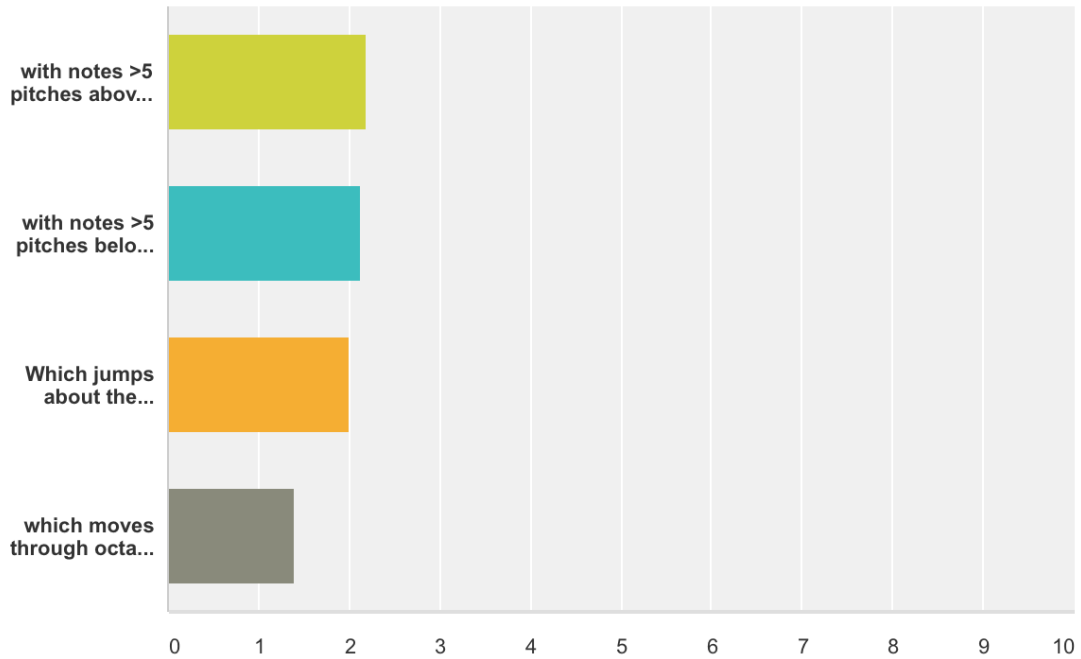


	Easy	Medium	Hard	Total	Weighted Average
with a tempo of > than 120BPM	26.67% 8	50.00% 15	23.33% 7	30	1.97
with a tempo of < than 40 BPM	50.00% 15	36.67% 11	13.33% 4	30	1.63

[Comments](#) (8)

## Difficulty of a piece...

Answered: 30 Skipped: 7

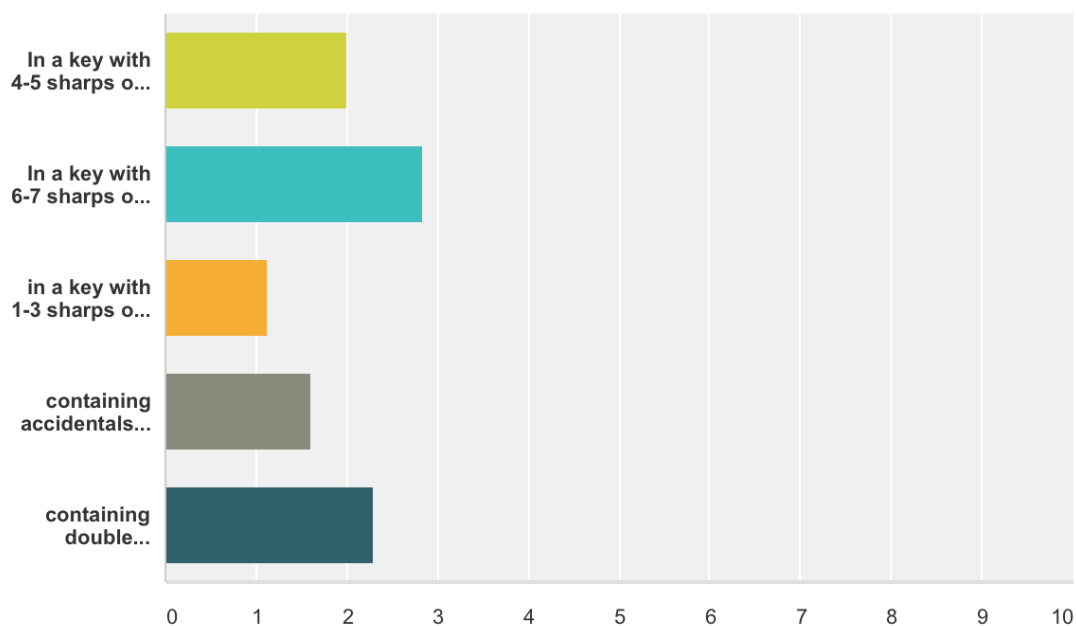


	Easy	Medium	Hard	Total	Weighted Average
with notes >5 pitches above the stave	20.00% 6	40.00% 12	40.00% 12	30	2.20
with notes >5 pitches below the stave	26.67% 8	33.33% 10	40.00% 12	30	2.13
Which jumps about the octaves	20.00% 6	60.00% 18	20.00% 6	30	2.00
which moves through octaves more sequentially	63.33% 19	33.33% 10	3.33% 1	30	1.40

[Comments \(6\)](#)

## Difficulty of a piece...

Answered: 30 Skipped: 7



	Easy	Medium	Hard	Total	Weighted Average
▼ In a key with 4-5 sharps or flats	13.33% 4	73.33% 22	13.33% 4	30	2.00
▼ In a key with 6-7 sharps or flats	3.33% 1	10.00% 3	86.67% 26	30	2.83
▼ in a key with 1-3 sharps or flats	90.00% 27	6.67% 2	3.33% 1	30	1.13
▼ containing accidentals (sharps/flats/naturals)	43.33% 13	53.33% 16	3.33% 1	30	1.60
▼ containing double sharps/flats	10.00% 3	50.00% 15	40.00% 12	30	2.30

[Comments \(6\)](#)

The final question provided a space for the participant to enter any details of their instrument which were not answered by any of the other questions.



**Finally, are there any other things you use to assess upon first view/play whether a piece is going to be difficult/easy, excluding things like "guest spot" or "beginner guest spot" or "playalong" being in the book title?**

Answered: 23 Skipped: 14

Responses (23)

Text Analysis

My Categories

#### PRO FEATURE

Use text analysis to search and categorize responses; see frequently-used words and phrases. To use Text Analysis, upgrade to a GOLD or PLATINUM plan.

Upgrade

[Learn more »](#)

Categorize as...

Filter by Category

Search responses

Showing 23 responses

Cue viola joke re: viola solo ----- Clef changes Not alot of different clefs are used anymore. (saw your tweet with loads of them) Violas generally play in the Alto clef so treble clef means high notes for a while. Its a bit harder as it is high. Our fingers are closer so harder to be in tune. The body of the instrument gets in the way of the left hand so you have to be a bit contorted. Pushing the string down is harder as in the middle of the string. Also clef changes can be missed and are a little bit jarring. (Not as bad as going from C++ to C, as alot of viola players start on the violin.) We'd play the violin if we wanted to play high treble clef notes ! -----

--- Base 2 Things split/joined into multiples of 2 are nicer. eg.rhythms, bowing etc eg. playing 8 quavers in a 4/4 bar bowing : 8 in one direction - nice 4 down, 4 up - nice (2 down, 2 up) x 2 - nice 1,1x4 - nice 6 (4+2) down 2 up - ok but can run out of bow if repeated 5d 3up - not so nice 7d 1 up - not nice Same for rhythms -----

- Music Markings positions / fingering markings bowing markings - up, down, a number of notes in one direction If there are lots of these, it tells you how to play and generally means it works well as the composer know what they are doing. ----- More advance techniques pizzicato (plucked string) / spiccato, marccato, martele jete bowed / col legno harmonics glissando sul tasto / ponticello ----- Changes The more things change, the harder it is both mentally and also physically. eg. frequent clef changes you can practise until you can play. hand positions and shapes - take time to change pizzicato (plucked string) / spiccato, marccato, martele bowed / col legno take longer to change May become impossible to actually play at speed. =====

Composer's ability A competent string player can easily play a piece written by someone who knows the instrument well even if it jumps about the octaves, goes high, is fast, has lots of short notes, lots of accidentals, pizzicato, changing dynamics, changing bowing, strange keys etc. A "simpler" piece written by someone who doesn't know the instrument or is doing more modern music can be very hard even for a very good player as left hand position and shape can make it almost impossible to play or hurt when doing it . see box 9 example.

## D User guide

Listing 2: key signature testcase

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 2.0 Partwise//EN" "http://www.
  musicxml.org/dtds/partwise.dtd">
3 <score-partwise>
4   <identification>
5     <encoding>
6       <software>MuseScore 1.3</software>
7       <encoding-date>2014-12-22</encoding-date>

```

```

8      </encoding>
9      </identification>
10     <defaults>
11       <scaling>
12         <millimeters>7.05556</millimeters>
13         <tenths>40</tenths>
14       </scaling>
15     <page-layout>
16       <page-height>1683.78</page-height>
17       <page-width>1190.55</page-width>
18       <page-margins type="even">
19         <left-margin>56.6929</left-margin>
20         <right-margin>56.6929</right-margin>
21         <top-margin>56.6929</top-margin>
22         <bottom-margin>113.386</bottom-margin>
23       </page-margins>
24       <page-margins type="odd">
25         <left-margin>56.6929</left-margin>
26         <right-margin>56.6929</right-margin>
27         <top-margin>56.6929</top-margin>
28         <bottom-margin>113.386</bottom-margin>
29       </page-margins>
30     </page-layout>
31   </defaults>
32   <part-list>
33     <score-part id="P1">
34       <part-name>Flute</part-name>
35       <part-abbreviation>Fl.</part-abbreviation>
36       <score-instrument id="P1-I3">
37         <instrument-name>Flute</instrument-name>
38       </score-instrument>
39       <midi-instrument id="P1-I3">
40         <midi-channel>1</midi-channel>
41         <midi-program>74</midi-program>
42         <volume>78.7402</volume>
43         <pan>0</pan>
44       </midi-instrument>
45     </score-part>
46   </part-list>
47   <part id="P1">
48     <measure number="1" width="158.18">
49       <print>
50         <system-layout>
51           <system-margins>
52             <left-margin>65.78</left-margin>
53             <right-margin>0.00</right-margin>
54           </system-margins>
55           <top-system-distance>70.00</top-system-distance>
56         </system-layout>
57       </print>
58     <attributes>
59       <divisions>1</divisions>
60       <key>
61         <fifths>1</fifths>
62         <mode>major</mode>
63       </key>
64       <time>
65         <beats>4</beats>
66         <beat-type>4</beat-type>
67       </time>
68       <clef>
69         <sign>G</sign>
70         <line>2</line>
71       </clef>
72     </attributes>
73     <note>
74       <rest/>
75       <duration>4</duration>

```

```

76     <voice>1</voice>
77     </note>
78   </measure>
79   <measure number="2" width="116.43">
80     <attributes>
81       <key>
82         <fifths>2</fifths>
83         <mode>major</mode>
84       </key>
85     </attributes>
86     <note>
87       <rest/>
88       <duration>4</duration>
89       <voice>1</voice>
90     </note>
91   </measure>
92   <measure number="3" width="126.43">
93     <attributes>
94       <key>
95         <fifths>3</fifths>
96         <mode>major</mode>
97       </key>
98     </attributes>
99     <note>
100       <rest/>
101       <duration>4</duration>
102       <voice>1</voice>
103     </note>
104   </measure>
105   <measure number="4" width="136.43">
106     <attributes>
107       <key>
108         <fifths>4</fifths>
109         <mode>major</mode>
110       </key>
111     </attributes>
112     <note>
113       <rest/>
114       <duration>4</duration>
115       <voice>1</voice>
116     </note>
117   </measure>
118   <measure number="5" width="146.43">
119     <attributes>
120       <key>
121         <fifths>5</fifths>
122         <mode>major</mode>
123       </key>
124     </attributes>
125     <note>
126       <rest/>
127       <duration>4</duration>
128       <voice>1</voice>
129     </note>
130   </measure>
131   <measure number="6" width="156.43">
132     <attributes>
133       <key>
134         <fifths>6</fifths>
135         <mode>major</mode>
136       </key>
137     </attributes>
138     <note>
139       <rest/>
140       <duration>4</duration>
141       <voice>1</voice>
142     </note>
143   </measure>

```

```

144 <measure number="7" width="171.03">
145   <attributes>
146     <key>
147       <fifths>7</fifths>
148       <mode>major</mode>
149     </key>
150   </attributes>
151   <note>
152     <rest/>
153     <duration>4</duration>
154     <voice>1</voice>
155   </note>
156   <barline location="right">
157     <bar-style>light-light</bar-style>
158   </barline>
159 </measure>
160 <measure number="8" width="243.24">
161   <print new-system="yes">
162     <system-layout>
163       <system-margins>
164         <left-margin>42.50</left-margin>
165         <right-margin>0.00</right-margin>
166       </system-margins>
167       <system-distance>92.50</system-distance>
168     </system-layout>
169   </print>
170   <attributes>
171     <key>
172       <fifths>-7</fifths>
173       <mode>major</mode>
174     </key>
175   </attributes>
176   <note>
177     <rest/>
178     <duration>4</duration>
179     <voice>1</voice>
180   </note>
181 </measure>
182 <measure number="9" width="142.59">
183   <attributes>
184     <key>
185       <fifths>-6</fifths>
186       <mode>major</mode>
187     </key>
188   </attributes>
189   <note>
190     <rest/>
191     <duration>4</duration>
192     <voice>1</voice>
193   </note>
194 </measure>
195 <measure number="10" width="132.59">
196   <attributes>
197     <key>
198       <fifths>-5</fifths>
199       <mode>major</mode>
200     </key>
201   </attributes>
202   <note>
203     <rest/>
204     <duration>4</duration>
205     <voice>1</voice>
206   </note>
207 </measure>
208 <measure number="11" width="122.59">
209   <attributes>
210     <key>
211       <fifths>-4</fifths>

```

```

212         <mode>major</mode>
213     </key>
214 </attributes>
215 <note>
216     <rest/>
217     <duration>4</duration>
218     <voice>1</voice>
219 </note>
220 </measure>
221 <measure number="12" width="112.59">
222     <attributes>
223         <key>
224             <fifths>-3</fifths>
225             <mode>major</mode>
226         </key>
227     </attributes>
228     <note>
229         <rest/>
230         <duration>4</duration>
231         <voice>1</voice>
232     </note>
233 </measure>
234 <measure number="13" width="102.59">
235     <attributes>
236         <key>
237             <fifths>-2</fifths>
238             <mode>major</mode>
239         </key>
240     </attributes>
241     <note>
242         <rest/>
243         <duration>4</duration>
244         <voice>1</voice>
245     </note>
246 </measure>
247 <measure number="14" width="92.59">
248     <attributes>
249         <key>
250             <fifths>-1</fifths>
251             <mode>major</mode>
252         </key>
253     </attributes>
254     <note>
255         <rest/>
256         <duration>4</duration>
257         <voice>1</voice>
258     </note>
259 </measure>
260 <measure number="15" width="85.86">
261     <attributes>
262         <key>
263             <fifths>0</fifths>
264             <mode>major</mode>
265         </key>
266     </attributes>
267     <note>
268         <rest/>
269         <duration>4</duration>
270         <voice>1</voice>
271     </note>
272     <barline location="right">
273         <bar-style>light-light</bar-style>
274     </barline>
275 </measure>
276 </part>
277 </score-partwise>

```

## G References